Introduction aux blockchains

Responsable du cours : Xavier Goaoc

Mode d'emploi

Commençons par préciser de quelle manière il convient d'aborder ce cours et ce polycopié.

Objectifs pédagogiques.

Les « chaine de blocs »(« *blockchain* »en anglais) sont des systèmes numériques qui visent à **construire de la confiance** entre des acteurs qui ne se connaissent pas. Elles ambitionnent de se substituer à diverses structures de nos sociétés humaines : certification de chaine logistique, enregistrement de contrat, tenue de cadastre, système monétaire, ...

Techniquement, une chaine de blocs est un système d'enregistrement d'information, un *registre*, qui combine deux propriétés. D'une part, ce registre est **décentralisé** au sens où il est entretenu et mis à jour conjointement par un ensemble d'acteurs indépendants les uns des autres. D'autre part, ce registre est **infalsifiable** au sens où il est facile pour chaque acteur de déterminer si une copie donnée du registre a été altérée.

La conception d'une chaine de blocs met en jeu de la **cryptographie** et du **calcul distribué**. Différents choix techniques (algorithmes, fonctions mathématiques, ...) conduisent à différents types de chaines de blocs et, *in fine*, induisent différents sens aux mots "acteurs indépendants" et "infalsifiable". Appréhender le sens de ces adjectifs dans une chaine de bloc donnée demande d'examiner ses principes de fonctionnement. Ce cours a pour objectif de vous donner les clefs pour amorcer une telle analyse.

Organisation.

Le cours commence par trois séances de cryptographie dédiées au **hachage** et à la **signature**. Ensuite, vient une séance posant le cadre du **calcul distribué** et plus précisément les problèmes de l'**élection** et du **consensus**. Avec ces notions à notre disposition, on peut alors procéder à une **étude de cas** sur le **bitcoin**, plus précisément son élection par **preuve de travail**, son **consensus probabiliste**, et son **coût écologique désastreux**[1] pour une **capacité très limitée**[2]. Suit une dernière séance de calcul distribué dévolue aux algorithmes de consensus, puis la séance d'examen.

Le polycopié entremêle notes de cours et exercices. La difficulté des exercices de TD est mesurée par l'échelle (approximative) suivante :

★	Ne demande aucune connaissance préalable ou simple vérification de la compréhension d'une définition.
★★	Application directe d'une idée supposée acquise ou expliquée en cours.
★★★	Demande une idée originale, ou une application originale d'une idée supposée acquise ou expliquée en cours, ou une application combinée de plusieurs idées supposées acquises ou expliquées en cours.

1. En 2018, cette chaine de blocs a consommé de l'ordre de 30 terawatt-heures, soit un coût moyen de 380 kilowatt-heures *par transaction*. Pour fixer les idées, 380 kilowatt-heures correspond à la quantité d'énergie qu'il faut apporter à $\sim 4m^3$ d'eau pour faire passer, dans des conditions standard, ce volume d'eau de 20 à 100 degrés Celsius.

2. En 2018, cette chaine de blocs a enregistré $\sim$ 81 millions de transactions. Cet ordre de grandeur du nombre de transactions est intrinsèquement limité par la conception de cette chaine de blocs. Pour fixer les idées, réserver l'intégralité de leur usage à la Communauté Urbaine du Grand Nancy ne permettrait même pas une transaction par personne par jour.

Évaluation

L'évaluation du cours repose pour partie sur les **rendus hebdomadaires** du travail de TD, pour partie sur un **examen écrit final** portant sur l'analyse d'un système réel de chaine de blocs à partir des *white papers* décrivant sa conception.

> Le seul document autorisé à l'examen est la **copie papier** du polycopié, qui peut être annotée. Aucun système électronique ne sera autorisé, pas même pour vous donner l'heure.

Remerciements

Je tiens à remercier Pierrick Gaudry, Eve-Angéline Lambert, Olivier Mirgaux, Cyril Nicaud, Rémi Peyre et Bogdan Warinschi pour leurs conseils, suggestions et explications qui m'ont beaucoup aidé à préparer ce cours. Les erreurs, coquilles et autres approximations sont bien évidemment de ma responsabilité.

Table des matières

Chapitre 1

Construire un registre inaltérable par hachage cryptographique

Cette séance décrit comment on peut utiliser des *fonctions de hachage* pour réaliser un registre informatique *inaltérable*. L'objectif n'est pas seulement de comprendre cette construction (qui est très simple), mais d'apprendre à jauger la confiance que l'on peut avoir en ce caractère inaltérable.

Les objectifs sont que vous...

- compreniez la méthodologie cryptographique, qui fait reposer la sécurité de *protocoles* sur certaines propriétés de *primitives cryptographiques* pas toujours formellement prouvées, mais expérimentalement éprouvées,
- appreniez à apprécier les limites des méthodes « par force brute » que sont l'énumération déterministe et l'exploration aléatoire.

1.1 Principe d'un registre inaltérable

Le terme de *registre informatique* désigne de manière générale un système informatique qui permet de sauvegarder une séquence de « blocs de données ». Comme un registre papier que l'on remplit page après page, potentiellement sur une longue durée, un registre informatique permet d'ajouter des blocs de données au fil du temps. Le premier enjeu des chaine de blocs est de construire des registres informatiques dont on peut *facilement* vérifier l'authenticité d'une copie.

1.1.1 Terminologie : mot binaire et bloc de données

On appelle *mot binaire* un élément de $\bigcup_{n\in\mathbb{N}^*}\{0,1\}^n$, c'est à dire une suite *finie* de chiffres, chacun valant 0 ou 1. Chacun de ces chiffres binaires est appelé *bit*, contraction de *bi*nary dig*it*.

La très grande majorité des ordinateurs actuels sont numériques, c'est à dire qu'ils manipulent l'information sous forme de mots binaires. Ainsi, en première approximation, lorsqu'un ordinateur manipule une information (texte, image, son, vidéo, programme informatique, etc.), il la représente [1] par un mot binaire. Les détails de cet « encodage en mot binaire » nous importent généralement peu : on va s'abstraire de cette représentation et travailler directement sur des mots binaires.

Il est courant de diviser un mot binaire de taille arbitraire en une séquence de mots 8 bits. [2] Cette pratique a pour origine l'ingéniérie des systèmes d'information, dont les unités mémoire travaillent par unité de mot 8 bits. Un mot binaire de 8 bits est appelé *octet*. Dans ce cours, on appelera *bloc de données* une suite d'octets qui représente (la division d')un mot binaire.

1. La procédure de conversion d'un objet informatique en un mot binaire est appelé *sérialisation*.
2. Si la longueur du bloc n'est pas un multiple de 8, on considère que le dernier octet est complété par une convention quelconque.

Si w est un mot binaire, on note $|w|$ son nombre de bits et $\text{bin}(w)$ le nombre entier ayant w comme écriture binaire. Ainsi, $|100| = 3$ et $\text{bin}(100) = 4 = \text{bin}(0100)$. On note $\cdot$ l'opération de concaténation, ainsi $1001 \cdot 11 = 100111$.

La *notation hexadécimale* présente un nombre en base 16 avec pour chiffres $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$. Puisque $16 = 2^4$, la conversion entre binaire et hexadécimal peut se faire "sans mémoire", chaque chiffre hexadécimal correspondant exactement à un mot binaire 4 bits : $0 \leftrightarrow 0000$, $1 \leftrightarrow 0001$,..., $f \leftrightarrow 1111$. Cela mets en correspondance les mots binaires de longueur multiple de 4 avec les mots hexadécimaux, c'est à dire sur l'alphabet $\{0, 1, \ldots, f\}$. On présentera les mots binaires un peu longs, disons à partir de 8 bits, sous une telle notation hexadécimale, avec les conventions d'usage suivantes :

- Si la longueur du mot binaire n'est pas multiple de 4, on le complète à gauche par des 0 de manière à arrondir cette taille au multiple de 4 immédiatement supérieur. On laisse cette complétion (*padding* en anglais) implicite sauf si elle est susceptible d'avoir une incidence.
- On signalera les nombres écrits en hexadécimal au moyen du préfixe 0x ('zéro-x').
- On utilisera indifféremment minuscules et majuscules. Ainsi `0xcafe` = `0xCAFE` = `0xCaFe`.

1.1.2 Fonction de hachage

Une **fonction de hachage** est une fonction qui associe à tout mot binaire un mot binaire de taille fixée. Autrement dit, une fonction de hachage est une fonction de $\bigcup_{n\in\mathbb{N}^*}\{0,1\}^n$ dans $\{0,1\}^\ell$ pour un entier ℓ donné, que l'on appelle la *taille* de la fonction de hachage.

Voici trois exemples (deux triviaux, un utile) de fonctions de hachage :

- La fonction qui à un mot w associe 1 si w contient un nombre pair de 1 et 0 sinon est une fonction de hachage de taille 1.
- La fonction qui à un mot w associe le mot $u \in \{0,1\}^\ell$ tel que $\text{bin}(u) = \text{bin}(w) \mod 2^\ell$ est une fonction de hachage de taille ℓ.
- Fixons deux entiers $1 < a < p$ où p est premier et $p > 2^\ell$. La fonction $f_{a,p}$ qui associe à un mot w le mot $u \in \{0,1\}^\ell$ tel que $\text{bin}(u) = (a * \text{bin}(w) \mod p) \mod 2^\ell$ est une fonction de hachage de taille ℓ. Oui, c'est celle-ci qui peut être utile[3], mais pas dans le cadre de ce cours.

Une fonction de hachage est donc une manière d'associer à toute suite d'information une « empreinte » de taille fixée. L'image d'un mot binaire par une fonction de hachage est appelé le **haché** de ce mot (par cette fonction).

1.1.3 Principe de chainage de blocs

De manière informelle, une *chaine de blocs* (ou *blockchain* en anglais) est une séquence $(B_1, B_2, \ldots, B_n)$ de blocs de données telle que B_i contienne le haché de B_{i-1}, pour $2 \leq i \leq n$. Pour formaliser cette définition, il conviendrait d'expliciter deux points :

(i) Quelle est la fonction de hachage H utilisée ?

(ii) Où trouve-t-on $H(B_{i-1})$ dans B_i ? Par exemple, est-ce que ce sont les premier ℓ bits de B_i, où ℓ est la taille de H ?

Le point (i) est essentiel. Le point (ii) semble relever d'une « convention de codage », au même titre que le choix de la manière de représenter une image ou un son par un mot binaire. C'est une première

3. On peut montrer que pour tous mots binaires w, w', si l'on choisit a aléatoirement uniformément dans $\{2, 3, \ldots, p-1\}$, la probabilité que $f_{a,p}(w) = f_{a,p}(w')$ est $O(2^{-\ell})$. Autrement dit, une table de hachage basée sur une telle fonction de hachage (choisie aléatoirement une fois pour toute à la création de la table) garantit effectivement un temps d'accès *moyen* en $O(1)$ pour une séquence raisonnablement petite d'objets quelconques. Et non, une table de hachage ne permet pas de ranger n objets en garantissant un accès de complexité $O(1)$: cette complexité s'entendant comme le *pire-cas*, elle est $O(n)$ quel que soit la fonction de hachage. Cf `http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf` pour plus de plus amples détails (mais ceci sort du cadre de ce cours).

approximation raisonnable[4] et nous supposons dans ce cours que nous savons « extraire » $H(B_{i-1})$ de B_i, sans détailler comment.

La supposée inaltérabilité des chaînes de bloc tient à deux ingrédients. Le premier ingrédient est résumé par la propriété suivante (que l'on étendra légèrement en exercice).

Proposition 1. *Soient* $(B_1, B_2, \ldots, B_n)$ *et* $(C_1, C_2, \ldots, C_n)$ *deux chaînes de blocs distinctes*[5]. *Si* $H(B_n) = H(C_n)$ *alors il existe* $1 \leq i \leq n$ *tel que* $B_i \neq C_i$ *et* $H(B_i) = H(C_i)$.

Démonstration. Posons $j^* \stackrel{\text{def}}{=} \min\{j \in \mathbb{N} : B_{n-j} \neq C_{n-j}\}$. Un tel indice existe car les deux chaînes de blocs sont supposées distinctes. On a d'une part que $B_{n-j^*} \neq C_{n-j^*}$, par définition de j^*. On a d'autre part que $H(B_{n-j^*}) = H(C_{n-j^*})$. Si $j^* = 0$ c'est une hypothèse. Si $j^* > 0$, remarquons que $B_{n-j^*+1} = C_{n-j^*+1}$. Comme $H(B_{n-j^*})$ apparaît dans B_{n-j^*+1}, et idem pour les $C_\bullet$, il en découle que $H(B_{n-j^*}) = H(C_{n-j^*})$. □

Supposons maintenant que l'on ait construit une chaîne de blocs $(B_1, B_2, \ldots, B_n)$ qui a été largement répliquée, et que l'on souhaite vérifier l'authenticité d'une copie $(C_1, C_2, \ldots, C_n)$ que l'on nous présente. D'après la Proposition 1, toute modification dans $C_\bullet$ par rapport à $B_\bullet$ *qui préserve* $H(C_n) = H(B_n)$ demande de produire une *collision* pour H, c'est à dire deux mots binaires $C_i \neq B_i$ de mêmes hachés.

Réciproquement, supposons que l'on dispose d'une fonction de hachage H pour laquelle il est « pratiquement infaisable » de produire une collision. On peut alors authentifier la copie $C_\bullet$ en (i) vérifiant que chaque bloc C_i contient bien $H(C_{i-1})$ pour $1 \leq i \leq n$, et (ii) vérifiant que $H(C_n) = H(B_n)$. L'intérêt de cette méthode est qu'elle ne demande de connaître de $B_\bullet$ que $H(B_n)$, soit une quantité d'information bien plus petite que l'intégralité de la chaîne.

En **résumé**, une fonction de hachage qui « résiste aux collisions », au sens où il est *pratiquement infaisable* de construire deux mots binaires de même haché, permet d'authentifier un bloc de données arbitrairement grand par une quantité d'information fixée (autant de bits que la taille de la fonction de hachage). La méthode du chaînage permet d'authentifier une série de blocs par l'empreinte du dernier d'entre eux.

1.1.4 Exercices

Exercice 1 ★ (Avec machine)

a. Utilisez les `bytearray` de Python pour afficher, sous la forme d'un mot hexadécimal, le mot binaire codant la chaîne de caractères "*cours d'introduction aux blockchaines*".

b. Sur quelle convention s'appuie ce codage d'une chaîne de caractères en un mot binaire ? Justifiez votre réponse par un lien vers la partie pertinente de la documentation officielle de Python.

Exercice 2 ★★ (Avec machine) Revenons sur la fonction de hachage donnée en exemple ci-dessus :

> *Fixons deux entiers* $1 < a < p$ *où* p *est premier et* $p > 2^\ell$. *La fonction* $f_{a,p}$ *qui associe à un mot* w *le mot* $u \in \{0,1\}^\ell$ *tel que* $bin(u) = (a * bin(w) \mod p) \mod 2^\ell$ *est une fonction de hachage de taille* ℓ.

On fixe $\ell = 16$, $p = 17977$ et[6] $a = \texttt{0xcafe}$.

4. Ce point cache en fait des enjeux assez subtils, enjeux que nous illustrerons par les « attaques par extension de longueur » en seconde séance.

5. C'est à dire qu'il existe $1 \leq i \leq n$ tel que $B_i \neq C_i$.

6. Le préfixe `0x` indique que le nombre qui suit est noté en hexadécimal. L'hexadécimal est un système numérique en base 16 qui utilise comme chiffres $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$. La base 16 est plus compacte que l'écriture décimale et a l'avantage que ses chiffres sont en correspondance avec les chiffres binaires : chaque chiffre hexadécimal correspond à 4 chiffres binaires (car $16 = 2^4$).

a. Déterminez $f_{a,p}(\texttt{0xdecafbad})$.

b. Déterminez un mot w formant avec `0xdecafbad` une collision pour $f_{a,p}$.

Exercice 3 ★ Indiquez pour chacune des affirmations suivantes si elle est vraie ou fausse. Justifiez chaque réponse par une preuve ou un contre-exemple.

a. Pour tout entier s, pour toute fonction de hachage H à valeurs dans $\{1, 2, \ldots, s\}$, pour tout mot binaire w, il existe un mot binaire w' tel que (w, w') est une collision pour H.

b. Pour tous entiers s_1, s_2, pour toutes fonctions de hachage H_1 et H_2 à valeurs, respectivement, dans $\{1, 2, \ldots, s_1\}$ et $\{1, 2, \ldots, s_2\}$, il existe une paire (w, w') de mots binaires qui est une collision pour H_1 et pour H_2.

c. Pour tout entier k, pour tous entiers s_1, s_2, ..., s_k, pour toutes fonctions de hachage H_i à valeurs dans $\{1, 2, \ldots, s_i\}$, $1 \leq i \leq k$, il existe une paire (w, w') de mots binaires qui est une collision pour *chacune* des H_i.

d. Pour tout entier s, pour toute fonction de hachage H à valeurs dans $\{1, 2, \ldots, s\}$, il existe un mot binaire w pour lequel il existe une infinité de mots binaires w' tels que (w, w') est une collision pour H.

1.2 Exemples d'utilisations de fonctions de hachage

Le principe de construction d'une chaine de blocs illustre une démarche courante en cryptographie : **réduire** la sécurité d'un système à certaines **propriétés clefs** de ses composants. Ainsi la Proposition 1 énonce que si l'on a confiance dans l'impossibilité de construire des collisions pour une fonction de hachage H, alors on peut avoir confiance dans l'impossibilité de modifier une chaine de blocs construite à partir de H sans modifier son haché terminal $H(B_n)$.

Voyons quelques exemples de systèmes informatiques que l'on peut construire à partir de fonctions de hachage, et dont la fiabilité se réduit à des propriétés cryptographiques de ces fonctions.

1.2.1 Le reader's digest

Supposons que l'on souhaite archiver quelques milliers de fichiers informatique, disons des images faisant chacune quelques méga-octets. On souhaite déterminer si un nouveau fichier `f` que l'on nous présente fait déjà partie de la liste. La question que l'on se pose n'est pas de savoir si on a déjà un fichier de ce nom là, mais si l'un de nos fichier contient la même image (*i.e.* est codée par le même mot binaire).

Ce problème, fondamental en gestion de données, est connu sous le nom de *problème du dictionnaire*. Dans certains cas, simplement comparer `f` à chacun des objets archivés peut s'avérer gourmand en ressources. Une première solution simple, mais déjà bien plus efficace que la comparaison naïve des fichiers, se base sur une fonction de hachage H ayant la propriété suivante :

> **Résistance à la collision.** Il est infaisable en pratique de calculer deux mots binaires w et w' tels que $H(w) = H(w')$.

On peut alors utiliser cette fonction H comme *empreinte* de nos objets. Si les objets archivés sont $A_1, A_2, \ldots$, on précalcule $H(A_1)$, $H(A_2)$, ... et on les stocke. Ensuite, à chaque objet A présenté, on calcule $H(A)$ et on le compare aux $H(A_1)$, $H(A_2)$, ..., ce qui ne demande que de travailler sur des résumés de la taille de H. Pour éviter les faux positifs, il suffit, en cas de correspondance entre deux hachés, de vérifier que les images sont effectivement les mêmes.[7]

7. En cas de faux positif avéré, on prévient les autorités que l'on a trouvé une collision pour H afin de la de déchoir de son statut « cryptographique ».

On retrouve cette idée dans les *tables de hachage*, mais avec une différence importante. Une table de hachage utilise une fonction de hachage de taille ℓ faible, de façon à pouvoir créer un tableau de taille 2^ℓ en mémoire. Cette fonction ne peut donc être résistante aux collisions, et les faux positifs sont inévitables. (C'est pourquoi une table de hachage **ne** garantit **pas** un accès en temps $O(1)$ dans le pire-cas.[8])

1.2.2 Mots de passe

Comment un système informatique peut-il faire pour être capable de *vérifier* le mot de passe d'un utilisateur sans pour autant le stocker de manière trop accessible ? Supposons que l'on dispose d'une fonction de hachage H satisfaisant l'hypothèse suivante :

> **Résistance à la première pré-image.** Étant donné un haché h, il est infaisable en pratique de calculer un mot binaire w tel que $H(w) = h$.

On peut alors se dispenser de stocker les mots de passe, et ne stocker que leurs hachés par H. Lorsque quelqu'un prétend s'authentifier en tant qu'Alice au moyen d'un mot de passe m, on calcule $H(m)$ et on le compare au haché h du mot de passe d'Alice (que l'on a stocké). Si $H(m)$ égale h alors on accepte le mot de passe, sinon on le refuse.

Remarquons que dans ce système (mis en œuvre par exemple par `passwd` sur unix), tant qu'il est difficile de calculer une pré-image pour H, il est difficile à quelqu'un ayant accès aux données stockées (les hachés des mots de passe) d'usurper une identité.[9] Autrement dit, la propriété de sécurité du système « ne pas pouvoir usurper une identité quand on a accès à la liste des hachés » repose sur l'hypothèse « H est résistante à la première pré-image ». Remarquons que si les progrès de la cryptanalyse font que H cesse d'être résistante à la première pré-image, le système perd sa propriété de sécurité ; on peut cependant la restaurer en remplaçant H par une nouvelle fonction de hachage, elle résistante.[10]

1.2.3 L'engagement

Lors d'une enchère secrète (par exemple "au premier prix"), on doit s'engager sur une décision (par exemple une proposition de prix) tout en la gardant secrète. Dans le monde physique, cela peut se faire par exemple par enveloppes cachetées. Dans le monde numérique, cela peut se faire au moyen d'une fonction de hachage cryptographique H satisfaisant l'hypothèse suivante :

> **Résistance à la seconde pré-image.** Étant donné un mot binaire w, il est infaisable en pratique de calculer un mot binaire $w' \neq w$ tel que $H(w') = H(w)$.

De telles enchères sont réalisées en deux phases. Dans la première phase, chaque participant choisit un mot binaire qui décrit son offre, disons w_i pour le i-ème participant, et annonce $H(w_i)$ à tous les participants. Une fois que tous les participants ont annoncé leur $H(w_i)$, chacun d'entre eux annonce son w_i. La première phase a pour objectif d'engager chaque participant (changer son choix tout en restant cohérent avec la valeur $H(w_i)$ annoncée demande de calculer une seconde pré-image) tout en gardant son information secrète (pour les autres participants, déduire w_i de $H(w_i)$ nécessite au minimum de calculer une première pré-image).

Pour mettre en œuvre ce principe élégant de manière sûre, il convient d'examiner certaines vulnérabilités *indépendantes de la fonction de hachage*. Par exemple, si les choix à faire sont dans un espace trop petit (par exemple un prix en euros payable par une municipalité), on peut deviner w_i à partir

8. Autrement dit : dites-moi quelle fonction vous utilisez et je me fais fort de trouver une suite d'entrées d'objets qui seront tous rangés dans la même case de votre table...

9. On ne laisse pas pour autant ces hachés en accès libre ; par exemple, de nombreux systèmes linux les stockent dans fichier comme `/etc/shadow` dont l'accès demande des droits de super-utilisateur.

10. Ne pas oublier de demander à tous les utilisateurs de réinitialiser leur mot de passe avec un outil intégrant cette nouvelle fonction.

de $H(w_i)$ en calculant simplement $H(n)$ pour tous les nombres n de 1 à 10^9. Bien entendu, cela peut être résolu en demandant à chaque participant d'ajouter à sa propositions un mot binaire arbitraire de grande taille qui sera ignoré au moment du décodage. Mais cela n'est qu'un exemple de vulnérabilité...

1.2.4 Exercices

Exercice 4 ★★ Soient $(B_1, B_2, \ldots, B_n)$ et $(C_1, C_2, \ldots, C_m)$ deux chaînes de blocs de fonction de hachage H. On suppose que ces chaines ont même bloc initial fixé ($B_1 = C_1$) et même haché final ($H(B_n) = H(C_m)$). (En particulier, le contenu du haché du « bloc précédant C_1 » est fixé.)

a. Supposons qu'il existe $0 \leq j < n$ tel que $B_{n-j} \neq C_{m-j}$. Montrez qu'on peut produire une collision pour H.

b. Supposons maintenant que $n < m$ et pour tout $0 \leq j < n$ on a $B_{n-j} = C_{m-j}$. Quelle hypothèse cryptographique est mise en danger ? De quelle manière ?

c. Résumez les hypothèse cryptographiques sur H qui assurent la propriété suivante : *« si $B_1 = C_1$ et $H(B_n) = H(C_m)$ alors $n = m$ et $B_i = C_i$ pour tout $1 \leq i \leq n$ ».*

Exercice 5 ★★ Revenons sur les trois hypothèses cryptographiques que l'on a présenté : résistance aux collisions, résistance à la première/seconde préimage. Prouvez que l'une de ces hypothèse en implique une autre.

Exercice 6 ★★ Supposons que H_1 et H_2 soient deux fonctions de hachage.

a. On note $H_1 \circ H_2$ la composition. À quelle(s) condition(s) sur H_1 et H_2 est-ce que $H_1 \circ H_2$ est résistante aux collisions ?

b. On note $H_1 \cdot H_2$ la fonction obtenue par concaténation, c'est à dire que $H_1 \cdot H_2(x) = H_1(x) \cdot H_2(x)$. À quelle(s) condition(s) sur H_1 et H_2 est-ce que $H_1 \cdot H_2$ est résistante aux collisions ?

1.3 Fonctions de hachage cryptographiques

La méthodologie cryptographique réduit la confiance en certaines propriétés d'un système (par exemple un dispositif d'authentification) à certaines propriétés de primitives cryptographiques (par exemple la résistance à la première pré-image d'une fonction de hachage). Cette réduction rend essentielle une bonne appréciation de la confiance qu'il convient d'accorder dans le fait qu'une primitive cryptographique a bien les propriétés qu'on lui prête. C'est ce que nous allons maintenant examiner.

Une fonction de hachage est dite **cryptographique** si elle satisfait : (i) résistance à la collision, (ii) résistance à la première pré-image, et (iii) résistance à la seconde pré-image.

Rappelons que les conditions (i)–(iii) font intervenir l'expression vague « il est infaisable en pratique de calculer », aussi cette définition est pour l'instant au mieux une convention de langage. Voyons ce qui se cache derrière...

1.3.1 Il est délicat de prouver le caractère cryptographique

Pourrait-on **prouver** qu'une fonction de hachage donnée est résistante aux collisions, au sens où tout algorithme qui calcule une telle collision est de grande complexité ? La question n'est pas saugrenue : en effet, en informatique théorique, le domaine de la *théorie de la complexité* s'intéresse précisément à établir des bornes inférieures, éventuellement conditionnelles, sur la complexité de **tout algorithme** résolvant un problème donné.

Le principe des tiroirs assure que pour toute fonction de hachage H de taille ℓ, il existe deux mots w et w' de tailles au plus $\ell+1$ tels que $H(w) = H(w')$. Établir une borne inférieure sur la complexité d'un algorithme calculant une collision se heurte donc à un problème de taille : pour *toute* fonction de hachage H de taille ℓ, il existe un algorithme de complexité $O(\ell)$ qui retourne une collision ! Un tel algorithme est de la forme

```
fonction collision()
    renvoyer (a,b)
```

Bien évidemment, on ne connaît pas les valeurs a et b qui font que cet algorithme est correct pour une fonction H donnée. Il n'empêche, **il existe** un algorithme de cette forme qui est correct pour H. L'existence d'un tel algorithme rend la formalisation théorique de la résistance aux collisions délicate.

Une approche standard consiste à changer le problème. On peut par exemple considérer une *famille* de fonctions de hachage $\{H^s : s \in S\}$, où S est un ensemble de paramètres assez grand. Une telle famille *résiste à la collision* si étant donné une valeur du paramètre s, il est difficile de calculer une collision pour H^s. Cette formulation rend l'objection ci-dessus inopérante et autorise par exemple des preuves de NP-difficulté. En revanche, il n'est pas évident que ce nouveau problème **modélise** correctement notre souhait de sécurité. Autrement dit, il n'est pas évident qu'une fonction pour laquelle ce problème est prouvé difficile (par exemple NP-difficile) soit fiable d'un point de vue cryptographique : un usage frauduleux n'a besoin que de calculer quelques collisions, éventuellement bien choisies. Dans ce cours, nous n'approfondissons donc pas cette approche.

1.3.2 Mais alors, qu'est-ce qu'une « fonction de hachage cryptographique » ?

Une approche courante consiste à traiter le terme de « fonction de hachage cryptographique » non pas comme un terme technique, mais comme désignant un **consensus** de la communauté qui s'intéresse à ces questions de sécurité informatique. Le consensus en question est... qu'il est pratiquement infaisable de calculer une collision, une première pré-image ou une seconde pré-image. Cela a une conséquence immédiate :

> Le fait qu'une fonction de hachage soit considérée comme *cryptographique* n'est en rien définitif.

En effet, le consensus sur ce qui est pratiquement faisable ou infaisable peut changer. Nous illustrerons cela en section 1.3.6 après avoir examiné les difficultés *pratiques* de la recherche d'une collision.

1.3.3 Rendre l'énumération inopérante

L'hypothèse cryptographique ne porte pas sur l'existence d'une collision, mais bien sur la difficulté pratique d'en calculer une. L'idée qu'il soit difficile de trouver une collision pour une fonction de hachage H peut paraître surprenante puisqu'il suffit d'énumérer les mots binaires jusqu'à en trouver une, comme le fait par exemple l'algorithme suivant (en pseudo-code proche de Python).

```
1 fonction enumération(H)
2     vu = [(0,H(0))]
3     i, h = 1, H(1)
4     pour tout (a,b) dans vu:
5         si b == h:
6             renvoyer (i,a)
7     insérer (i,h) dans vu
8     i = i+1 puis h = H(i)
9     retourner à la ligne 4
```

Précisons qu'on identifie ici un entier et le mot formé par son écriture binaire minimale, c'est à dire sans 0 à gauche (sauf pour l'entier 0). Ainsi $H(42)$ vaut $H(101010)$.

Il se trouve que dès que la taille de la fonction de hachage est suffisamment grande, cette solution par énumération est inefficace en pratique. Pour cela, il peut être utile d'avoir à l'esprit les ordres de grandeur suivants [11] :

2^{35}	$\approx$	le nombre d'opérations en virgule flottante réalisées en une seconde par un processeur 8 cœurs à ≤ 4 GHz
2^{60}	$\approx$	le nombre d'opérations en virgule flottante réalisées en une seconde par le superordinateur *Frontier*
2^{16}	$\approx$	le nombre de secondes dans une journée
2^{30}	$\approx$	le nombre de secondes dans un siècle

Ainsi, simplement *compter* jusqu'à 2^{65} est une tâche difficile pour un ordinateur standard (il mettra un siècle) mais facile pour un supercalculateur (le plus rapide au moment où ce polycopié est rédigé le fait en moins d'une minute). Compter jusqu'à 2^{100} s'avère titanesque pour tout ordinateur actuel (le *Frontier* y passerait de nombreux siècles).

La technologie évoluant [12] il peut être utile de se donner quelques grandeurs « naturelles » :

2^{60}	$\geq$	le nombre de secondes restant avant que le Soleil d'absorbe la Terre
2^{240}	$\geq$	le nombre estimé d'atomes dans l'univers observable

Ainsi, une fonction de hachage de taille suffisante, disons au moins 256 bits, et dont les valeurs sont suffisamment distribuées pour qu'une répétition ne se produise pas trop vite « résistera » à une approche par énumération.

1.3.4 Rendre les anniversaires inopérants

On peut chercher une collision pour une fonction de hachage H par un algorithme probabiliste, suivant le principe suivant :

```
1 fonction alea(H)
2     vu=[] puis i=random() puis h=H(i)
3     si (i,h) n'est pas dans vu:
4         pour tout (a,b) dans vu:
5             si b == h:
6                 renvoyer (i,a)
7         insérer (i,h) dans vu
8     i = random() puis h = H(i)
9     retourner à la ligne 3
```

On suppose ici que la fonction `random()` retourne une chaîne binaire aléatoire choisie uniformément dans $\{0,1\}^{\ell'}$, où ℓ' est strictement supérieur à la taille ℓ de H. Quelle valeur de ℓ faut-il choisir pour que cette approche soit inefficace en pratique ?

Modélisation probabiliste

Posons $N \stackrel{\text{def}}{=} 2^\ell$ et notons $[N] \stackrel{\text{def}}{=} \{1, 2, \ldots, N\}$. Notons $\mathcal{U}$ la loi uniforme sur $[N]$. Notons μ la loi de probabilité sur $[N]$ définie par

$$\mu(i) \stackrel{\text{def}}{=} \mathbb{P}\left[\text{bin}(H(x)) = i - 1\right]$$

pour $i \in [N]$ et x une variable aléatoire uniforme dans $\{0,1\}^{\ell'}$. Considérons une suite $h_1, h_2, \ldots$ de variables aléatoires indépendantes de lois μ et définissons

- $p(\mu, N, k)$ la probabilité que les valeurs $h_1, h_2, \ldots, h_k$ soient deux à deux distinctes,

11. Pour les conversions décimal-binaire, utiliser $2^{10} = 1024 \simeq 10^3$ et donc $2^k \simeq 10^{k/3}$.
12. Le *Frontier* est devenu le superordinateur le plus rapide du monde en 2022.

- $c(\mu, N) \stackrel{\text{def}}{=} \min\{i : \exists 1 \leq j < i \text{ t.q. } h_j = h_i\}$ l'indice[13] de première collision.

L'algorithme `alea` est d'autant moins efficace que ces quantités sont grandes. L'analyse qui suit établit que ces quantités sont maximales lorsque $\mu = \mathcal{U}$ et que pour $\mathcal{U}$, la première collision se produit typiquement vers $\approx \sqrt{N}$ tirages. Cela permet de tirer les conclusions suivantes :

> Une fonction de hachage résiste d'autant mieux à une recherche « par anniversaire » que ses images sont bien réparties sur $\{0,1\}^\ell$.

> Pour résister à une recherche « par anniversaire » il est nécessaire que $2^{\ell/2}$ soit grand au sens de la Section 1.3.3.

Analyse probabiliste

Notons $\binom{[N]}{k}$ l'ensemble des sous-ensembles de $[N]$ de taille k. Pour $X \in \binom{[N]}{k}$ notons $p(\mu, N, X) \stackrel{\text{def}}{=} \prod_{x \in X} \mu(x)$. En sommant les probabilités des tirages (ordonnés) sans collision on obtient

$$p(\mu, N, k) = k! \sum_{X \in \binom{[N]}{k}} p(\mu, N, X). \tag{1.1}$$

Par ailleurs, les variables p et c sont liées par $\mathbb{P}[c(\mu, N) > k] = p(\mu, N, k)$, et donc

$$\mathbb{E}[c(\mu, N)] = \sum_{k=1}^{N} k\, \mathbb{P}[c(\mu, N) = k] = \sum_{k=1}^{N} \mathbb{P}[c(\mu, N) \geq k] = \sum_{t=2}^{N-1} p(\mu, N, t). \tag{1.2}$$

Commençons par établir la domination de la loi uniforme.

Lemme 2. *Pour tout N, pour toute mesure μ non-uniforme sur $[N]$,*

$$\mathbb{E}[c(\mathcal{U}, N)] > \mathbb{E}[c(\mu, N)] \quad \textit{et} \quad \forall 1 \leq k \leq N, \quad p(\mathcal{U}, N, k) > p(\mu, N, k).$$

Démonstration. Puisque $\mu \neq \mathcal{U}$ il existe $x, y \in \{1, 2, \ldots, N\}$ tels que $\mu(x) < \frac{1}{N} < \mu(y)$. Posons $\delta \stackrel{\text{def}}{=} \min(1/N - x, y - 1/N)$ et définissons une nouvelle mesure $\hat{\mu}$ sur $\{1, 2, \ldots, N\}$ par

$$\hat{\mu}(x) \stackrel{\text{def}}{=} \mu(x) + \delta, \quad \hat{\mu}(y) \stackrel{\text{def}}{=} \mu(y) - \delta, \quad \hat{\mu}_{|\{1,2,\ldots,N\}\setminus\{x,y\}} = \mu_{|\{1,2,\ldots,N\}\setminus\{x,y\}}.$$

Comparons terme à terme les reformulations de $p(\hat{\mu}, N, k)$ et $p(\mu, N, k)$ via la formule (1.1) :

- Pour les $X \in \binom{[N]}{k}$ qui ne contient ni x ni y on a $p(\hat{\mu}, N, X) = p(\mu, N, X)$ par définition.
- Les $X \in \binom{[N]}{k}$ qui contiennent exactement un élément de $\{x, y\}$ donnent lieu à des compensations par paires. En effet, pour tout $Y \in \binom{[N]\setminus\{x,y\}}{k-1}$ on a

$$\begin{aligned} p(\hat{\mu}, N, Y \cup \{x\}) + p(\hat{\mu}, N, Y \cup \{y\}) &= (\hat{\mu}(x) + \hat{\mu}(y))\, p(\hat{\mu}, N, Y) \\ &= (\mu(x) + \mu(y))\, p(\mu, N, Y) \\ &= p(\mu, N, Y \cup \{x\}) + p(\mu, N, Y \cup \{y\}). \end{aligned}$$

- Pour les $X \in \binom{[N]}{k}$ qui contiennent x et y, en posant $Y \stackrel{\text{def}}{=} X \setminus \{x, y\}$, on a

$$p(\hat{\mu}, N, X) = \hat{\mu}(x)\hat{\mu}(y)p(\hat{\mu}, N, Y) > \mu(x)\mu(y)p(\mu, N, Y) = p(\mu, N, X),$$

  l'inégalité venant du fait que $\delta \leq \frac{\mu(y)-\mu(x)}{2}$ assure que $\hat{\mu}(x)\hat{\mu}(y) > \mu(x)\mu(y)$

On a par conséquent que $p(\hat{\mu}, N, k) > p(\mu, N, k)$. Remarquons qu'il y a strictement plus d'entiers de mesure exactement $1/N$ pour $\hat{\mu}$ que pour μ. Une récurrence immédiate implique donc que pour toute mesure μ non-uniforme sur $[N]$ on a $p(\mathcal{U}, N, k) > p(\mu, N, k)$, et donc aussi $\mathbb{E}[c(\mathcal{U}, N)] > \mathbb{E}[c(\mu, N)]$ via la formule (1.2). □

13. C'est une variable aléatoire dont la loi dépend de N et de μ.

Encadrons ensuite la probabilité de non-collision :

Lemme 3. *Pour* $1 \leq k \leq \sqrt{2N}$ *on a* $1 - \dfrac{k(k-1)}{2N} \leq p(\mathcal{U}, N, k) \leq 1 - \dfrac{k(k-1)}{4N}$.

Démonstration. Notons ϵ_k l'événement où les valeurs $h_1, h_2, \ldots, h_k$, tirées uniformément dans $[N]$, sont deux à deux distinctes. Chaque événement $h_i = h_j$ est de probabilité $1/N$. En considérant l'union de ces événement, on a

$$1 - p(\mathcal{U}, N, k) = 1 - \mathbb{P}[\epsilon_k] \leq \binom{k}{2} \frac{1}{N} = \frac{k(k-1)}{2N},$$

ce qui prouve la minoration annoncée. D'autre part,

$$p(\mathcal{U}, N, k) = \mathbb{P}[\epsilon_k] = \underbrace{\mathbb{P}[\epsilon_k \mid \epsilon_{k-1}]}_{=1-\frac{k-1}{N}} \mathbb{P}[\epsilon_{k-1}] = \prod_{i=1}^{k-1} \left(1 - \frac{i}{N}\right).$$

Pour tout $x \in [0,1]$ on a $1 - x \leq e^{-x} \leq 1 - \frac{x}{2}$. On a ainsi

$$p(\mathcal{U}, N, k) = \prod_{i=1}^{k-1} \left(1 - \frac{i}{N}\right) \leq \prod_{i=1}^{k-1} e^{-\frac{i}{N}} = e^{-\frac{k(k-1)}{2N}} \leq 1 - \frac{k(k-1)}{4N},$$

la majoration annoncée. □

Le Lemme 3 implique que pour $k = o(\sqrt{N})$, la probabilité qu'il n'y ait aucune collision est proche de 1, tandis que pour $k = \sqrt{2N}$, cette probabilité est au plus $\frac{1}{2}$. En particulier, la probabilité de n'avoir aucune collision après $t\sqrt{2N}$ tirages est au plus 2^{-t}, ce qui assure que pour la loi uniforme, la première collision se produit avec forte probabilité, vers $\Theta(\sqrt{N})$ tirages. Une analyse plus fine (et plus technique) révèle que $\mathbb{E}[c(\mathcal{U}, N)] = (1 + o(1))\sqrt{\frac{\pi N}{2}}$.

1.3.5 Miser sur l'effet d'avalanche

On peut définir et tester différents critères indiquant qu'une fonction de hachage est difficile à prédire. En voici un :

> *Notons T l'ensemble des paires (w, w') de mots ℓ-bits qui different en exactement un bit. On dit qu'une fonction de hachage H satisfait le critère d'avalanche strict si pour tout $0 \leq i < \ell$, lorsque l'on choisit (w, w') aléatoirement uniformément dans T, la probabilité que les $i$ème bits de $f(w)$ et de $f(w')$ soient égaux est $\frac{1}{2}$.*

Il s'agit là d'un critère théorique difficile à vérifier rigoureusement, mais qui peut être testé en échantillonnant T.

1.3.6 Ci-git SHA-1

Concluons par un exemple de « cycle de vie » d'une fonction de hachage utilisée en cryptographie. La fonction de hachage SHA-1 a été conçue par la NSA et rendue publique en 1995. Sa taille est de 160 bits, aussi une recherche de collision probabiliste devrait demander le calcul de $\sim 2^{80}$ hachés, ce qui était considéré comme suffisant à l'époque (mais ne satisfait plus aux exigences typiques d'aujourd'hui). Cette fonction a été largement déployée.

En 2005, SHA-1 a été déclarée "peu sûre contre des adversaires dotés de gros moyens" suite à la publication d'une attaque améliorée ($\sim 2^{69}$ calculs), puis d'une seconde ($\sim 2^{63}$). En février 2017, une collision a été exhibée. En janvier 2020, une paires de clefs PGP/GnuPG distinctes et de même haché SHA-1 ont été produites. Encore récemment, SHA-1 était utilisée pour des raisons de rétro-compatibilité ; en toute logique, cela devrait cesser rapidement. Nous renvoyons à l'introduction de l'article de Leurant et Peyrin [1] sur ce sujet.

1.3.7 Exercices

Exercice 7 ★★ (Avec machine) La librairie `hashlib` de Python [14] fournit des implémentation de calcul de fonctions de hachage cryptographiques. Utilisez cette librairie pour...

a. Calculer le haché par la fonction SHA-256 du mot “blockchaine”.

b. Trouver une chaîne de caractère de préfixe “les blockchaines c’est” et dont le haché a une écriture hexadécimale qui commence par 3 ou moins.

Exercice 8 ★★★ (Avec machine, optionnel) Testez expérimentalement le critère d’avalanche sur la fonction SHA-256.

1.4 Références bibliographiques

[1] Gaëtan Leurent and Thomas Peyrin. Sha-1 is a shambles : First chosen-prefix collision on sha-1 and application to the {PGP} web of trust. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1839–1856, 2020.

Notes personnelles

14. https://docs.python.org/3/library/hashlib.html

Chapitre 2

Regard technique sur le hachage cryptographique

Cette séance approfondit l'usage des fonction de hachage cryptographique, sous l'angle de la technique informatique. L'objectif est d'appréhender une fonction de hachage cryptographique comme un objet technique concret dont certaines propriétés sont subtiles. Pour cela, on présente en détail :

- la *construction de Merkle-Darmgård* sur laquelle sont basées plusieurs fonctions de hachage cryptographiques standard,
- la fonction SHA-256, construite par le principe de Merkle-Darmgård,
- *l'attaque par extension de longueur*, une faille de sécurité affectant certains usages de fonctions de hachages obtenues par la construction de Merkle-Darmgård, et
- *les arbres de Merkle*, des structures de données informatiques qui constituent la base du stockage d'information dans les chaines de blocs.

Les objectifs sont que vous...

- sachiez mener une réduction formelle de sécurité simple (comme à l'exercice 2) et une analyse de complexité d'une structure de donnée à base de pointeur hachés (comme à l'exercice 4),
- ayiez « disséqué » une fonction de hachage cryptographique largement utilisée (SHA-256),
- situiez la sécurité cryptographique dans un contexte plus large, qui autorise des attaques n'invalidant pas les propriétés cryptographiques (ici, l'attaque par extension de longueur),

2.1 Construction de Merkle-Darmgård

La *construction de Merkle-Darmgård* définit une fonction de hachage à partir d'une fonction opérant sur des mots binaires de longueur fixée. Elle a la propriété de transférer la résistance aux collisions de la seconde vers la première.

2.1.1 Définition

Fixons deux entiers ℓ et m, appelés, respectivement, *longueur de chaînage* et *longueur de bloc*. Fixons aussi une fonction $f : \{0,1\}^\ell \times \{0,1\}^m \to \{0,1\}^\ell$, appelée *fonction de compression* puisqu'elle transforme un mot binaire de longueur $m + \ell$ en un mot binaire de longueur ℓ.

La *construction de Merkle-Darmgård* définit une fonction de hachage g à partir de f et d'un mot ℓ-bits h_init, appelé *vecteur d'initialisation*. L'image $g(w)$ d'un mot binaire w fini, mais de longueur arbitraire, est définie en deux étapes.

La première étape est la complétion (« padding ») du mot w en un mot $\hat{w}$ dont la taille est multiple de m. La manière de réaliser cette complétion est laissée libre dans la mesure où elle satisfait les propriétés suivantes :

(c1) w est un préfixe de $\hat{w}$ (autrement dit : on complète par la fin),
(c2) si $|w_1| = |w_2|$ alors $|\hat{w}_1| = |\hat{w}_2|$, et
(c3) si $|w_1| \neq |w_2|$ alors les derniers m bits de $\hat{w}_1$ sont différents des derniers m bits de $\hat{w}_2$.

Nous verrons à l'exercice 1 un exemple de fonction de complétion, `shapad`.

Notons $\hat{w}$ le complété de w et décomposons le en $\hat{w} = w[0] \cdot w[1] \cdot \ldots \cdot w[n]$ où chaque $w[i]$ est un mot sur m bits. On itère ensuite f par

$$h_0 \stackrel{\text{def}}{=} h\_init \quad \text{et pour } 0 \leq i \leq n, \quad h_{i+1} \stackrel{\text{def}}{=} f(h_i, w[i]),$$

et la valeur h_{n+1} est ce que l'on définit comme $g(w)$. En pseudo-code, cela donne :

```
fonction g(w)
    h = h_init
    pour tout i de 1 à n+1:
        h = f(h,w[i])
    renvoyer h
```

Le choix de la règle de complétion, du mot initial et de la fonction de compression constituent des paramètres de la construction MD, que l'on retrouve par exemple dans l'ancienne fonction SHA-1 mais aussi dans l'actuelle fonction de hachage cryptographique SHA-256.

2.1.2 Propriété MD

L'intérêt de la construction MD est la propriété suivante (que l'on prouve à l'exercice 2) :

Théorème 4. *Si f est résistante aux collisions, alors g est résistante aux collisions.*

Autrement dit, pour construire une fonction de hachage résistant aux collisions sur des mots de taille arbitraire, il suffit d'en construire une qui résiste aux collisions sur des mots de taille fixée. Inversement, pour mettre à l'épreuve la résistance aux collisions d'une fonction de hachage produite par la construction MD, on peut se concentrer sur la fonction de compression sous-jacente.

2.1.3 Exercices

Exercice 1 ★ Fixons $m = 512$ et $\ell = 256$, et définissons la fonction de complétion `shapad` comme suit. Cette fonction prend en entrée un mot binaire w de longueur inférieure[1] à 2^{64}. Soit k le plus petit entier solution de $|w| + 1 + k = 448 \mod 512$. Pour tout entier $0 \leq x < 2^{64}$, notons $\text{bin}_{64}(x)$ l'encodage binaire de x sur 64 bits. On définit

$$\texttt{shapad}(w) \stackrel{\text{def}}{=} w \cdot 10^k \cdot \text{bin}_{64}(|w|)$$

où $\cdot$ désigne la concaténation. Remarquons que $|\texttt{shapad}(w)|$ est un multiple de 512.

a. Montrez que `shapad` satisfait les propriétés (c1), (c2) et (c3).
b. Peut-on déduire le mot w du mot $\texttt{shapad}(w)$?
c. Peut-on déduire la longueur $|\texttt{shapad}(w)|$ de la longueur $|w|$?
d. Montrez qu'aucune fonction de complétion définie sur des mots binaires de taille *arbitraire* ne peut satisfaire les propriétés (c1), (c2) et (c3).[2]

Exercice 2 ★★ Nous allons maintenant prouver le théorème 4. Considérons une fonction de hachage g obtenue par la construction MD à partir d'une fonction de compression f. Supposons que l'on connaisse $w_1 \neq w_2$ tels que $g(w_1) = g(w_2)$.

a. Montrez qu'il est facile de construire une collision pour f si $|w_1| \neq |w_2|$.
b. Montrez qu'il est facile de construire une collision pour f si $|w_1| = |w_2|$.

1. On s'éloigne ici légèrement de la définition que l'on a donné au chapitre 1, mais la nuance a peu d'importance en pratique : on se limite aux mots binaires occupant au plus 100 millions de téraoctets.
2. Autrement dit, l'approximation pointée à la note de bas de page précédente est nécessaire.

2.2 SHA-256

Sha-256 (aussi appelée SHA2-256) est une fonction de hachage considérée comme cryptographique obtenue par la construction de Merkle-Darmgård. Elle utilise $m = 512$ comme longueur de bloc et $\ell = 256$ comme longueur de chainage. Elle utilise `shapad`, vue à l'exercice 1, comme fonction de complétion ; en particulier, Sha-256 ne peut s'appliquer « qu'à » des mots binaires de taille inférieure strictement à 2^{64}.

On présente ci-dessous de manière explicite le vecteur d'initialisation et la fonction de compression utilisées par Sha-256. C'est une tâche assez laborieuse, sans grand intérêt conceptuel, qu'il s'agit d'aborder comme une opération de démontage d'un moteur en mécanique automobile : même quand on connaît les principes généraux de fonctionnement, il est parfois utile pour notre propre édification d'examiner un exemple réel de près.

2.2.1 Notations et opérations binaires

La fonction Sha-256 opère sur les mots binaires de longueur 256 ou 512 bits au travers de variables contenant chacune un mot binaire 32 bits. Sur ces mots 32 bits, on définit plusieurs opérations. La plus simple est $+$, qui désigne l'addition modulo 2^{32}. Les opérateurs booléens classiques (`et`, `ou`, `non`, `ou exclusif`) sont étendus aux mots binaires par les deux conventions suivantes :

- $0 \leftrightarrow$ `faux` et $1 \leftrightarrow$ `vrai`, et
- les opérations sont réalisées *bit à bit*.

On note les opérateurs ainsi obtenus par $\wedge$ (pour `et`), $\vee$ (pour `ou`), $\neg$ (pour `non`) et $\oplus$ (pour `ou exclusif`). Ainsi,

$$1001 \wedge 1100 = 1000, \quad 1001 \vee 1100 = 1101, \quad \neg 1001 = 0110, \quad \text{et} \quad 1001 \oplus 1100 = 0101.$$

On note aussi S_n l'opération consistant à décaler un mot binaire n fois vers la droite ; à chaque décalage, le bit le plus à droite est perdu et un 0 est ajouté à gauche pour garder la longueur constante. On note enfin R_n l'opération consistant à décaler un mot binaire *circulairement* n fois vers la droite. Ainsi,

$$S_2(1001) = 0010 \quad \text{et} \quad R_2(1001) = 0110.$$

2.2.2 Constantes

Le *vecteur d'initialisation* h_init est fixé à

```
h_init = 0x6a09e667bb67ae853c6ef372a54ff53a510e527f9b05688c1f83d9ab5be0cd19
```

Par ailleurs, la fonction de compression de Sha-256 utilise 64 constantes 32 bits, notées K_0 à K_{63}, fixés (dans l'ordre) à

```
0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2
```

2.2.3 Chargement du chainage et du bloc

Le premier argument de f est un mot de taille $\ell = 256$ bits. Il est chargé dans 8 variables 32 bits, notées de a à h, a représentant la partie de poids fort. Ainsi, par exemple, au premier appel de f dans la construction MD, ces variables sont initialisées à :

$$h\_init = \underbrace{6a09e667}_{a}\,\underbrace{bb67ae85}_{b}\,\underbrace{3c6ef372}_{c}\,\underbrace{a54ff53a}_{d}\,\underbrace{510e527f}_{e}\,\underbrace{9b05688c}_{f}\,\underbrace{1f83d9ab}_{g}\,\underbrace{5be0cd19}_{h}.$$

On crée 8 variables a' à h', que l'on initialise par des copies des valeurs ainsi chargées dans a à h. Ces copies nous servirons à la dernière étape du calcul de la fonction de compression (qui va modifier les variables a à h).

Le second argument de f est un mot w' de taille $m = 512$ bits. Il est chargé dans 16 variables 32 bits W_0 à W_{15}, de sorte que

$$w' = W_0 \cdot W_1 \cdot \ldots \cdot W_{15}$$

($\cdot$ étant l'opération de concaténation). Une fois W_0 à W_{15} ainsi initialisés, on définit 48 autres variables, nommées W_{16} à W_{63}, par

$$W_j \stackrel{\text{def}}{=} \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$$

où

$$\sigma_0(x) \stackrel{\text{def}}{=} S_7(x) \oplus S_{18}(x) \oplus R_3(x) \quad \text{et} \quad \sigma_1(x) \stackrel{\text{def}}{=} S_{17}(x) \oplus S_{19}(x) \oplus R_{10}(x).$$

2.2.4 La fonction de calcul

Une fois le chainage et le bloc chargés, la fonction de compression itère les opérations suivantes pour j allant de 0 à 63 :

- Calculer $\alpha \stackrel{\text{def}}{=} (e \wedge f) \oplus (\neg e \wedge g)$, $\beta \stackrel{\text{def}}{=} (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$, $\gamma \stackrel{\text{def}}{=} S_2(a) \oplus S_{13}(a) \oplus S_{22}(a)$ et $\delta \stackrel{\text{def}}{=} S_6(e) \oplus S_{11}(e) \oplus S_{25}(e)$.
- Calculer $T_1 \stackrel{\text{def}}{=} h + \alpha + \delta + K_j + W_j$ et $T_2 \stackrel{\text{def}}{=} \beta + \gamma$.
- Mettre à jour, dans l'ordre,

$$h \leftarrow g, \quad g \leftarrow f, \quad f \leftarrow e, \quad e \leftarrow d + T_1, \quad d \leftarrow c, \quad c \leftarrow b, \quad b \leftarrow a, \quad a \leftarrow T_1 + T_2.$$

À l'issu de ces 64 itérations, on retourne le mot 256 bits

$$(a + a') \cdot (b + b') \cdot (c + c') \cdot (d + d') \cdot (e + e') \cdot (f + f') \cdot (g + g') \cdot (h + h').$$

2.2.5 Exercice

Exercice 3 ★ On suppose que l'on charge les registres a à h par le vecteur d'initialisation h_init. Que vaut $\alpha \stackrel{\text{def}}{=} (e \wedge f) \oplus (\neg e \wedge g)$? Vous pouvez par exemple utiliser les « bitwise operators » de python :

https://wiki.python.org/moin/BitwiseOperators

Exercice 4 ★★ Considérons les deux codes suivants, tirés de la documentation de la bibliothèque `hashlib` de Python, et qui produisent le même haché :

```
1 m = hashlib.sha256()
2 m.update(b"Nobody inspects")
3 m.update(b" the spammish repetition")
```

```
1 m = hashlib.sha256()
2 m.update(b"Nobody inspects the spammish repetition")
```

Pensez-vous que la ligne 3 du premier code recalcule le haché à partir de zéro, et fait donc autant de calcul que la ligne 2 du second code, ou qu'au contraire elle peut exploiter le fait que l'on a déjà calculé le haché du début de la chaine ? Argumentez votre réponse en vous réferrant à la construction de Merkle-Darmgård.

2.2.6 À propos de la cryptanalyse de SHA-256

Une première question est de savoir si SHA-256 satisfait bien aux exigences que le calcul de collision, de première préimage et de seconde préimage sont difficiles en pratique. Comme discuté au chapitre 1, il n'existe aucune fonction de hachage pour lesquelles on sait *prouver* ces propriétés (en ces termes). La *confiance* dans le fait qu'une fonction de hachage *résiste en pratique* se base donc sur des éléments indirects :

- SHA-256 est largement utilisée, aussi des ressources importantes (temps de recherche de spécialistes, moyens de calcul, etc.) sont consacrées à la recherche d'une collision. Un aperçu des résultats de ces recherches est visible à

  https://en.wikipedia.org/wiki/SHA-2#Cryptanalysis_and_validation

- Le test expérimental du critère d'avalanche strict sur SHA-256 s'avère encourageant.

2.3 Attaque par extension de longueur

Certains systèmes informatiques utilisent des fonctions de hachage cryptographique pour assurer certaines propriétés de la sécurité. L'idée sous-jacente est de conditionner[3] certaines propriétés exprimant la sécurité du système (par exemple le fait que l'on ne puisse pas usurper l'identité d'un utilisateur) aux propriétés de résistance aux collisions, à la première préimage ou à la seconde préimage de la fonction de hachage. Il s'avère parfois que la sécurité du système peut être compromise *sans* que l'on n'ait à compromettre la nature cryptographique de la fonction. Un exemple bien connu est la vulnérabilité des utilisations de fonctions de hachages obtenues par la construction MD aux *attaques par extension de longueur*.

2.3.1 L'idée

Le principe de l'attaque par extension de longueur est le suivant :

> Pour tous mots binaires w_1 et w_2, il est facile de calculer SHA-256($\texttt{shapad}(w_1) \cdot w_2$) à partir de SHA-256($w_1$), de $|w_1|$ et de w_2.

En particulier, il n'est pas utile de connaître w_1 et on peut choisir w_2. Remarquons que si l'on savait calculer w_1, cela compromettrait le caractère cryptographique de SHA-256 puisque l'on aurait réussi à calculer une première préimage. L'attaque par extension de longueur contourne cette difficulté.

2.3.2 Le comment

Notons f la fonction de compression de SHA-256. Soit w_1 un mot binaire inconnu. On suppose que l'on connaît SHA-256(w_1) et la longueur $|w_1|$. Fixons un mot binaire w_2 arbitraire, et posons

$$w_2' \stackrel{\text{def}}{=} \texttt{shapad}(w_1) \cdot w_2.$$

Notons $w_3 \stackrel{\text{def}}{=} \texttt{shapad}(w_2')$. Comme chaque mot est préfixe de son complété (propriété c1), w_3 s'écrit

$$w_3 = w_2' \cdot s = \texttt{shapad}(w_1) \cdot w_2 \cdot s$$

pour un certain mot binaire s (un *s*uffixe). Remarquons que pour calculer s, la fonction `shapad` ne requiert que la connaissance de la longueur $|w_2'| = |\texttt{shapad}(w_1)| + |w_2|$. On connaît $|w_2|$ puisqu'on a choisi ce mot, et on peut facilement déduire la longueur $|\texttt{shapad}(w_1)|$ de la longueur $|w_1|$ (cf exercice 1). On peut donc *facilement* calculer s.

Notons maintenant $w_3 = w[0] \cdot w[1] \cdot \ldots \cdot w[n]$ une décomposition de w_3 en blocs de longueur 512. Comme $\texttt{shapad}(w_1)$ est un préfixe de m_3 et est de longueur un multiple de 512, il existe un indice

3. Un tel conditionnement peut parfois se prouver formellement. C'est un des champs d'application du domaine de recherche des *méthodes formelles*.

$0 \leq t \leq n$ tels que $\texttt{shapad}(w_1) = w[0] \cdot w[1] \cdot \ldots \cdot w[t]$. On ne connaît pas ces blocs, mais on peut en revanche *facilement* calculer les suivants :

$$w[t+1] \cdot w[t+2] \cdot \ldots \cdot w[n] = w_2 \cdot s.$$

Rappelons que la construction MD définit $g(w_2')$ comme le terme h_{n+1} de la suite

$$h_0 \stackrel{\text{def}}{=} h\_init \quad \text{et pour } 0 \leq i \leq n, \quad h_{i+1} \stackrel{\text{def}}{=} f(h_i, w[i]),$$

On connaît h_0 mais pas $w[0]$, aussi on ne sait pas calculer h_1, h_2, etc. En revanche, remarquons que h_{t+1} n'est autre que $g(w_1)$. On peut donc commencer à calculer les termes de cette suite « en chemin » et atteindre ainsi $h_{n+1} = g(w_2')$:

```
fonction ext()
    h = g(w1)
    pour tout i de t+1 à n
        h = f(h,w[i])
    renvoyer h
```

2.3.3 Le pourquoi : protocole MAC

Les *Message Authentification Codes* (MAC) sont des systèmes visant à permettre à deux interlocuteurs échangeant des messages sur un canal ouvert de vérifier que le message qu'ils reçoivent a bien été envoyé par l'autre interlocuteur, et non pas dû à une interférence d'une troisième personne (présumée mal-intentionnée). L'enjeu ici n'est pas d'empêcher quelqu'un d'avoir connaissance du message transmis, mais bien de certifier que le message reçu a bien été émis *tel quel* par l'autre interlocuteur.

Un système MAC simple consiste à ce que les interlocuteurs se mettent d'accord sur une fonction de hachage cryptographique h (par exemple SHA-256) et un secret commun. Ce secret commun est un mot binaire s qu'ils connaissent tous les deux et ne divulguent à personne d'autre. Lorsque l'un des interlocuteurs souhaite envoyer un mot binaire w à l'autre, il lui ajoute une signature qui est facile à calculer quand on connaît s, mais difficile sinon ; par exemple, il transmet $(w, h(w \cdot s))$. À la réception d'une paire (w', t), l'interlocuteur calcule $h(w' \cdot s)$ et accepte w' comme authentique si et seulement si $t = h(s \cdot w')$.

Ce système a été utilisé, par exemple, dans les API web de services comme `Flikr`. Sans entrer dans les détails, un utilisateur souhaitant faire certaines opérations sur son album photo déclenchait un échange de message authentifiés ainsi entre son navigateur et un serveur `Flikr`. Outre l'authentification, ces messages contenait des commandes à exécuter et leurs arguments.

Ce système de MAC par hachage d'un secret partagé permet beaucoup de souplesse. On peut, par exemple, convenir que l'on hachera non pas $w \cdot s$ mais $s \cdot w$. Ces choix semblent équivalent, mais ce simple détail entrouvre une faille de sécurité lorsque la fonction de hachage utilisée est basée sur la construction de MD (et, pour simplifier ici, `shapad`). En effet, remarquons que l'on dispose du mot w et du haché $h(s \cdot w)$. Si l'on connaît la longueur de s (et peu de tentatives devraient suffire), on peut en déduire la longueur $|s \cdot w|$ et ainsi calculer, par extension de longueur, le haché $h(\texttt{shapad}(s \cdot w) \cdot w')$ pour n'importe quel mot w'. Remarquons que $\texttt{shapad}(s \cdot w) = s \cdot w \cdot u$ pour un certain mot binaire u, que l'on peut calculer à partir de $|s \cdot w|$.

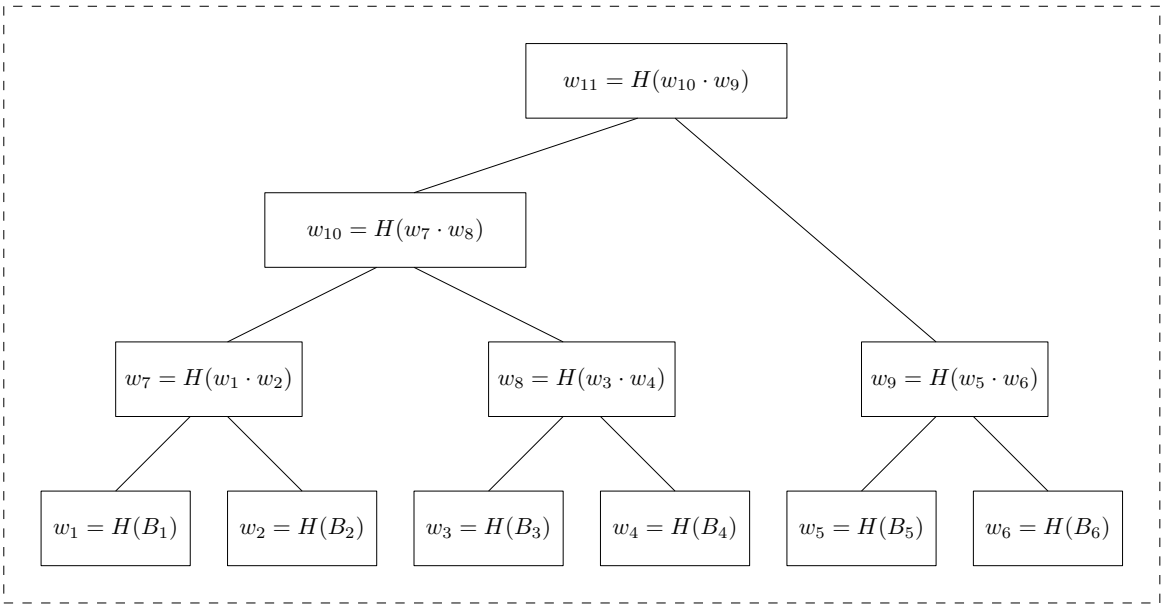
Comment serait interprété un message $(w \cdot u \cdot w', h(\texttt{shapad}(s \cdot w) \cdot w')$? D'une part, il serait reconnu comme authentique et donc ne serait pas rejetté. D'autre part, la commande $w \cdot u \cdot w'$ a de grandes chances de contenir des arguments aberrants de par la présence du terme u, non contrôlé ; certains analyseurs syntaxiques traitent de telles valeurs aberrantes en les ignorant *elles seules*, et en exécutant le reste de la commande. Dans une telle situation, les arguments passés par le mot w' que l'on a choisi sont *de facto* pris en compte.

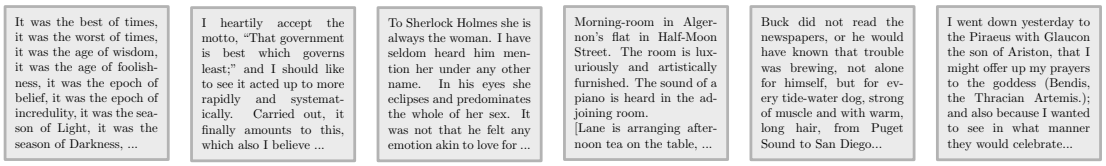
Cela peut sembler beaucoup de conditions... mais il s'est trouvé un certain nombre de services web pour y être vulnérables, dont `Flikr`. Vous trouverez plus de détails sur les failles de sécurités ouvertes par cette « attaque par extension de longueur » par exemple dans la notification de vulnérabilité :

2.4 Arbres de hachage

Terminons ce chapitre par l'examen d'une structure de donnée à base de fonctions de hachage : les *arbres binaires de hachage*, aussi appelés *arbres de Merkle*. On décrit cette structure abstraitement, du point de vue de l'information qu'elle organise, et on laisse de côté le détail de son implémentation.

Fixons une fonction de hachage H. Un arbre de Merkle est un arbre binaire qui référence un ensemble de blocs de données B_1, B_2, ... chaque bloc étant associé à une feuille. Chaque nœud N de l'arbre contient un mot binaire que l'on note w_N. Pour une feuille F, ce mot binaire est $w_F \stackrel{\text{def}}{=} H(B_F)$ le haché par H du bloc de donnée B_F associé à la feuille (le bloc de donnée ne fait pas partie de l'arbre, seulement son haché). Pour un nœud interne N de descendants N_1 et N_2, $w_N \stackrel{\text{def}}{=} H\left(w_{N_1} \cdot w_{N_2}\right)$ est le haché de la concaténation des mots de ses descendants. En voici une illustration ci-après.





B_1 B_2 B_3 B_4 B_5 B_6

Un arbre de Merkle permet de vérifier l'intégrité d'un ensemble de blocs dès lors que l'on a confiance en le mot de la racine. En effet, on peut prouver qu'il est facile de déterminer une collision pour H si l'on dispose de deux arbres de Merkle distincts dont les racines ont même mot binaire. Cette preuve, laissée en exercice, est similaire à la preuve de la Proposition 1.

Un arbre de Merkle présente en outre plusieurs avantages algorithmiques comparé aux listes chainées hachées. À titre d'exemples, citons :

- Cette structure occupe moins de place mémoire puisqu'elle ne contient pas les blocs de données référencés.
- Cette structure supporte la suppression de blocs. Il suffit d'effacer les feuilles correspondantes et, récursivement, tout nœud dont le parent n'a pas de petit-descendant (cf l'illustration ci-après).
- Pour une structure référençant n blocs, on peut définir un *certificat* de taille $O(\log n)$ permettant de vérifier l'appartenance d'un bloc. Il suffit pour cela de retracer le chemin de la racine de l'arbre à la feuille où est supposé rangé le bloc, et de fournir les mots de tous des nœuds et de leurs descendants hors du chemin.

Autrement dit, ces arbres permettent d'enregistrer les hachés d'un ensemble de blocs de manière à en permettre efficacement l'authentification et la suppression d'un bloc. Il existe des variantes qui permettent, par exemple, de fournir un certificat de taille $O(\log n)$ qu'un bloc n'est *pas* référencé.

2.4.1 Exercices

Exercice 5 ★ On souhaite vérifier l'intégrité d'un bloc de donnée stockée dans une structure S à base de hachage. On n'a pas confiance dans l'intégrité de la structure mais on est sûr du pointeur global, dont on avait gardé une copie en sécurité.

a. Comment fait-on la vérification dans le cas où S est une liste chaînée hachée ?

b. Comment fait-on la vérification dans le cas où S est un arbre de Merkle ?

c. On suppose que les structures sont optimisée : la liste ne contient pas de maillon "vide" et l'arbre de Merkle est complet. Combien de fonctions de hachage faut-il calculer au maximum sachant que S référence n blocs de données ?

Exercice 6 ★★ Supposons que l'on dispose de deux arbres de Merkle $\mathcal{A}$ et $\mathcal{A}'$ construits à partir d'une même fonction de hachage H. On suppose que les mots associés aux racines de $\mathcal{A}$ et $\mathcal{A}'$ sont égaux, mais que $\mathcal{A}$ et $\mathcal{A}'$ sont différents au sens où il n'existe pas de bijection envoyant les nœuds de $\mathcal{A}$ sur ceux de $\mathcal{A}'$ en préservant à la fois les relation de parenté et les mots binaires associés. Donnez une méthode pour construire une collision pour H.

Exercice 7 ★★ Supposons que deux machines aient stocké des copies de deux séquences de blocs de données. On souhaite vérifier que ces séquences sont identique, c'est à dire que les deux copies sont cohérentes. Proposez une solution efficace pour réaliser cela à base d'arbres de Merkle.

Exercice 8 ★★★ Proposez une variante des arbres de Merkle qui permette de fournir un certificat aussi efficace que possible qu'un bloc de donnée *n'est pas* référencé.

Notes personnelles

Chapitre 3

Identité cryptographique et preuve sans divulgation de connaissance

La séance 1 a esquissé les principes d'un registre inaltérable. La séance 4 amorcera sa décentralisation, c'est à dire l'organisation de sa tenue par un ensemble d'acteurs indépendants, pas nécessairement coopératifs, le tout sans coordination centrale. Cette séance prépare cette décentralisation en abordant les méthodes d'*identification* des acteurs. Cette identification se fait au moyen de *signatures cryptographiques* (« *digital signatures* ») ; nous en décrivons ici trois (signature de Schnorr, DSA et ECDSA) et esquissons leur utilisation pour la tenue d'un registre centralisé de transactions. Ces signatures sont baties sur l'idée de *preuve sans divulgation de connaissance* (« *zero knowledge proof* »), que l'on détaille aussi.

Les objectifs sont que vous...

- compreniez les principes d'une signature cryptographique « à la Schnorr » et un scenario d'usage de type cryptomonnaie,
- soyiez sensibilisé·e aux enjeux de sécurité sous-jacents (problème de logarithme discret, attaque sur générateurs pseudo-aléatoires) et en mesure d'interprêter des préconisation de type ANSSI.

3.1 Préambule : problématique de monnaie numérique

Commençons par examiner les principes généraux de construction d'une monnaie numérique, de manière à préciser ce que l'on attend d'un système de signature numérique.

3.1.1 Contexte : monnaie (numérique)

Une *monnaie* est un système qui remplit trois fonctions : c'est une *unité de compte* (elle permet de mesurer toute richesse), c'est un *intermédiaire des échanges* (tout ce qui s'achete ou se vend peut l'être contre une quantité de monnaie), et c'est une *réserve de valeur* (c'est une richesse en soi).

Un *instrument monétaire* est le support d'une unité de monnaie. Pièces et billets sont des instruments monétaires. Chèques, reconnaissances de dettes, bons de réduction, actions d'entreprise, *...n'en sont pas.* Ce sont des marchandises *portant sur* de l'argent, mais n'ayant pas *fonction* d'argent. La *valeur nominale* d'un instrument monétaire est la quantité de monnaie qu'il représente. Sa *valeur intrinsèque* est la quantité de monnaie équivalent à sa fabrication. Si ces deux quantités coincident, on parle de monnaie *de commodité*, sinon on parle de monnaie *fiduciaire*.

La *création monétaire* est l'acte de créer des instruments monétaires. Dans le cas d'une monnaie de commodité, la création de nouveaux instruments monétaires est limitée par la disponibilité de l'objet physique la réalisant : pour faire des Louis d'or, il faut de l'or. Dans le cas des instruments monétaires

physiques d'une monnaie comme l'Euro, cela est réalisé par une combinaison de difficulté techniques (filigrane, encres visibles aux infrarouges ou ultraviolets, hologrammes, etc.) et de dissuasion [1].

Une *transaction* est un acte transférant la propriété d'un instrument monétaire d'une personne à une autre. On dit que cet instrument a été *dépensé* par son ancien propriétaire. Lorsqu'un instrument monétaire est un objet physique, la *possession* de l'objet physique vaut possession de l'unité de monnaie correspondante ; *dépenser* une unité de monnaie c'est perdre possession de l'objet physique qui en est le support ; ce mécanisme empêche de dépenser une même unité de monnaie plusieurs fois.

Une monnaie est dite *numérique* (ou *digitale*) si ses instruments monétaires sont des mots binaires. Puisqu'un mot binaire est *réplicable à volonté*, toute monnaie numérique doit définir des mécanismes spécifiques de régulation de la création monétaire et de prévention de la *double-dépense* d'un instrument monétaire.

3.1.2 Registre et signature

On peut envisager de construire une « monnaie dématérialisée » rudimentaire [2] au moyen d'un registre. Pour cela, associons à chaque utilisateur un identifiant et numérotons les instruments monétaires. Le registre est constitué de blocs, chaque bloc enregistrant soit la création d'un instrument monétaire, soit une transaction (modifiant la propriété d'un instrument monétaire). C'est essentiellement ce que fait une banque. [3]

Une telle construction suppose de pouvoir attester qu'un utilisateur donne son accord à une transaction qui représente la dépense d'un instrument dont il est propriétaire. Dans le cas d'un registre tenu par une banque, cela est généralement réalisé par la *signature* d'un document papier our par l'utilisation d'un code *secret* pour valider une transaction.

Dans l'usage courant, une signature est une marque apposée par une personne (la ou le *signataire*) sur un *document*. Une signature *engage* au sens où elle établit que le signataire approuve le document. Pour cela, une signature doit avoir les propriétés suivantes :

- *Vérifiabilité* : tout le monde peut vérifier que la signature correspond au signataire.
- *Inforgeabilité* : seul le signataire peut apposer sa signature sur un document donné.
- *Spécificité* : une signature porte sur un seul document (elle de peut pas être « copiée et collée » sur un autre document.).

Les sceaux, tampons et signatures manuscrites ont pour objectif de réaliser cela.

Une **signature numérique** est un protocole qui permet de signer des mots binaires en assurant les propriétés de vérifiabilité, d'inforgeabilité et de spécificité. Ces signatures définissent l'identité comme *la connaissance d'un secret*, de manière similaire à l'usage des codes secrets de carte bancaire.

Une signature numérique permet donc de signer des textes, des images, des bases de données, . . .Les propriétés d'une signature se déduisent de propriétés de primitive cryptographique par un mécanisme de réduction similaire à celui utilisé pour prouver le Théorème 4.

1. Citons l'article 442-1 du code pénal Français : « *La contrefaçon ou la falsification des pièces de monnaie ou des billets de banque ayant cours légal en France ou émis par les institutions étrangères ou internationales habilitées à cette fin est punie de trente ans de réclusion criminelle et de 450 000 euros d'amende.* ». À titre de comparaison, l'artice 222-24 stipule que « *Le viol est puni de vingt ans de réclusion criminelle (1) Lorsqu'il a entraîné une mutilation ou une infirmité permanente, (2) Lorsqu'il est commis sur un mineur de quinze ans,. . .* ».
2. Cette « monnaie dématérialisée » est insatisfaisante à bien des égards mais va nous permettre d'introduire certaines des idées utiles à l'application Bitcoin, que l'on détaillera au chapitre suivant.
3. La tenue de compte par une banque ne constitue pas une monnaie mais cela en partage certaines caractéristiques.

3.1.3 Une monnaie numérique (centralisée)

On peut maintenant préciser un peu l'esquisse de monnaie dématérialisée de la Section 3.1.2. Alice pourrait réaliser une telle monnaie par un registre *public* listant, par ordre chronologique, des opérations. Les opérations possibles sont de deux types :

- `création` d'un instrument monétaire (disons une *pièce*). Un bloc création contient deux sous-parties. La partie (I) comporte l'identifiant de la pièce (un numéro), son montant (toutes les pièce ne se valent pas nécessairement), et l'identifiant du propriétaire. La partie (II) comporte la signature de la partie (I) par Alice
- `dépense/re-création`. Cette opération consomme des pièces et en crée de nouvelles. Un tel bloc contient deux sous-parties. La partie (I) comporte les identifiants d'une ou plusieurs pièces à consommer, ainsi que les identifiants et valeurs d'une ou plusieurs pièces à créer et, pour chacune, l'identifiant de leur propriétaire. La partie (II) comporte la signature de la partie (I) par *chaque* propriétaire d'une pièce consommée.

Alice fournit un pointeur chaîné sur le dernier bloc en cours (et le signe !). Les nouveaux blocs sont ajoutés par Alice, qui vérifie leur correction :

- le total des montants des pièces détruites doit est supérieur ou égal au total des pièces créés,
- chacune des pièces détruites doit avoir été créée et jamais détruite,
- chacun des propriétaires d'une pièce détruite doit avoir signé le bloc.

Ce système est *centralisé* : Alice contrôle la création de monnaie, garantit la bonne tenue des comptes, valide les nouvelles transactions... C'est assez similaire à la solution adoptée pour les dépots bancaires et comptes courants dans les banques commerciales[4] : ces problèmes sont réglés par la tenue d'un registre par la banque. Nous reviendrons dans les chapitres suivants sur la question de la *décentralisation* d'un tel système.

3.2 Principe d'une signature numérique

Comme annoncé ci-dessus, de nombreux systèmes de signature numérique *définissent* l'identité est comme la connaissance d'un secret. Cela implique qu'il doit être possible de vérifier la connaissance de ce secret (propriété de vérifiabilité) *sans qu'il ne soit révélé* (propriété d'inforgeabilité).

Examinons ces deux idées...

3.2.1 La connaissance d'un secret comme identité : exemple du chiffrement

Un procédé de **chiffrement** est un procédé d'encodage d'un mot binaire, appelé *message*[5], qui rend le décodage pratiquement très difficile à toute personne ne disposant pas d'une information, appelée *clef de déchiffrement*.[6] On peut matérialiser cette clef par un mot binaire indépendant du message, ce qui permet la formalisation suivante. Un procédé de chiffrement est une paire de fonctions $\mathrm{enc} : \{0,1\}^* \times K_e \to \{0,1\}^*$ et $\mathrm{dec} : \{0,1\}^* \times K_d \to \{0,1\}^*$, où K_e et K_d sont les sous-ensembles de $\{0,1\}^*$ de clefs de chiffrement et de déchiffrement. Des clefs $\alpha \in K_e$ et $\beta \in K_d$ sont dites *appariées* si

$$\forall w \in \{0,1\}^*, \quad \mathrm{dec}(\mathrm{enc}(w,\alpha),\beta) = w.$$

Autrement dit, on peut déchiffrer au moyen de la clef β ce qui a été chiffré au moyen de la clef α. De tels procédés peuvent par exemple être construits en interprétant les mots binaires comme des éléments d'une structure mathématique (par exemple un groupe) et en faisant agir, plus ou moins directement, la loi de cette structure sur le message et la clef.

Une méthode de chiffrement est *symmétrique* si les clefs de chiffrement et de déchiffrement sont identiques. Dans une méthode de chiffrement *asymmétrique*, chaque utilisateur dispose d'une paire

4. Ces dépots sont considérées comme une monnaie, dite *scripturale*, et elle est aujourd'hui numérique.

5. Bien entendu, cela s'applique aussi à des fichiers, des disques durs, etc. car ce ne sont que des exemples de mots binaires.

6. On ne s'intéresse pas, ici, aux mesures permettant d'éviter qu'un message chiffré ne soit intercepté. L'objectif est qu'un message chiffré reste indéchiffrable lorsqu'il tombe entre les mains de quelqu'un n'ayant pas la clef de déchiffrement.

de clefs appariées qui lui est propre. L'utilisateur diffuse largment la clef de chiffrement (appelée *clef publique*) et garde pour lui seul la clef de déchifrement (appellée *clef secrète*). Ainsi, toute entité ayant connaissance de la clef publique peut chiffrer un message à destination de l'utilisateur, que seul ce dernier est en mesure de déchiffrer.[7] Formellement, il est fréquent que l'appariement traduise une propriété mathématique reliant les deux clefs[8]. Ce lien rend possible *en principe* le calcul, étant donnée une clef de chiffrement α, d'une clef de déchiffrement β telle que (α, β) sont appariées. Un enjeu important de la sécurité de ces systèmes consiste à s'assurer que ce calcul soit *pratiquement infaisable*, au sens discuté en Séance 1.

3.2.2 Preuve sans divulgation de connaissance

Dans les méthodes de chiffrement décrites ci-dessus, on peut envisager les clefs comme des secrets dont la connaissance vaut droit de chiffrer ou déchiffrer. Dans le cas symmétrique, le secret doit être partagé par toutes les entités incluses dans la communication, et par elles seules. Dans le cas asymétrique, le secret donnant droit de chiffrer est largement diffusé, tandis que celui donnant droit de déchiffrer est détenu par une seule personne. Ces méthodes ont à leur cœur le principe suivant :

> La *connaissance* d'un secret identifie implicitement les personnes autorisées à déchiffrer un message.

On peut aller plus loin et *définir* explicitement l'identité comme la connaissance d'un secret. Cette définition s'avère utile en pratique grâce à l'idée suivante :

> Il est possible de prouver que l'on connaît un secret sans rien en révéler.

Illustrons cela sur un exemple concret. Comment Alice pourrait-elle prouver à Bob qu'elle connaît la solution à un sudoku sans qu'il ne puisse en déduire quoi que ce soit sur cette solution ? Cela peut se faire par le protocole suivant :

a. Alice et Bob tracent une grille 9×9 et préparent un 81 cartes de la taille d'une case ; chaque carte comporte au recto un numéro entre 1 et 9, neuf copies de chaque ; les cartes sont de verso indistinguables.

b. Alice et Bob conjointement posent sur chaque case connue dans *la donnée* du puzzle une carte de même numéro. Ces cartes sont posées faces visibles.

c. Ensuite, Alice distribue les cartes restantes, face cachée, sur les cases restantes et affirme que cette distribution forme une solution du sudoku.

d. Bob jette un dé à six faces. Si le résultat est 1 ou 2 il vérifie les lignes, si c'est 3 ou 4 il vérifie les colonnes, sinon il vérifie les blocs 3×3.

e. Alice et Bob retournent les cartes qui étaient faces visibles (les données du puzzle), puis regroupent les cartes faces cachées par paquets de 9, conformément au tirage de Bob.

f. Alice mélange chacun des paquets afin que Bob ne puisse pas déterminer à quelle case correspond chaque carte.

g. Bob vérifie que chaque paquet contient une carte de chacun des numéros de 1 à 9. Si la vérification échoue, cela prouve qu'Alice ne connaît pas la solution. Si la vérification réussit mais que Bob doute encore, ils recommencent en (b).

Soulignons qu'Alice s'engage *avant* que Bob ne décide (aléatoirement) de ce qu'il vérifie (lignes, colonnes ou blocs). Ainsi, si Alice ne connaît pas la solution au sudoku, sa disposition des cartes échoue au test pour au moins l'un des tirages possible. À chaque itération, Bob a donc une probabilité d'au moins $\frac{1}{3}$ de prendre Alice en défaut. Ces tests étant indépendants, la probabilité qu'Alice fasse illusion au cours de k répétitions est au plus $1/3^k$.

7. Dans un tel système, si on souhaite s'adresser à plusieurs utilisateurs il convient de chiffrer le message autant de fois qu'il y a de clefs publiques.

8. C'est ce qui explique que l'action de la clef de déchiffrement permette d'inverser l'action de la clef de chiffrement.

Pour avoir confiance dans le résultat de ce protocole, Bob n'a pas besoin d'avoir confiance en Alice, juste dans *sa propre* source d'aléa (son dé).

Remarquons que même si Alice passe le test, Bob n'a rien appris d'intéressant sur la solution.

Ce type de raisonnement peut se formaliser et se systématiser au moyen de *protocoles* entre *machines de Turing interactives* ayant accès à des sources d'aléa. Cette formalisation dépasse le cadre de ce cours, aussi on en donne seulement une esquisse en Annexe B pour les élèves intéressé·es. Ces principes sont cependant sous-jacents aux méthodes de signature que l'on va maintenant discuter.

3.2.3 Interface d'un système de signature

Les trois systèmes que l'on va examiner reposent sur l'utilisation d'une *clef*, c'est à dire une paire `(cp,cs)` formée de deux mots binaires. La *clef publique* `cp` est librement diffusée et permet à ceux qui la connaissent de *vérifier* une signature. La *clef secrète* `cs` permet à toute personne la connaissant de *signer* un mot binaire ; sa connaissance doit donc être limitée aux seules personnes supposées disposer de la signature (généralement, un seul individu).

Le système de signature fixe généralement les tailles de `cp` et `cs` à des constantes. Le système définit aussi quels sont les paires de mots binaires qui sont *appariés*, et forment donc une clef. Le système de signature est généralement constitué de trois fonctions :

- `genere_clef` ne prend pas d'argument et retourne une clef `(cp,cs)`. La clef retournée est choisie par la fonction de manière pseudo-aléatoire dans l'ensemble des paires de mots appariés.
- `signe` prend en arguments un message (sous la forme d'un mot binaire) et une clef secrète, et retourne un mot binaire que l'on appelle la *signature* du message par la clef.
- `verifie` prend en arguments un message, une signature et une clef publique, et décide si la clef publique donnée est appariée avec la clef secrète ayant produit la signature à partir du message.

Une large diffusion de la clef publique garantit la vérifiabilité du système. En revanche, son inforgeabilité repose sur le fait qu'il est difficile, étant donné une clef publique, de déterminer la (ou une) clef privée appariée.

3.2.4 Exercices

Exercice 1 ★★ (Preuve ZK) Proposez un protocole matériel (*i.e.* non numérique) permettant de réaliser des tâches suivantes. À chaque fois, Donnez l'idée du protocole et esquissez l'analyse de la probabilité qu'un menteur soit indétecté.

a. Étant donné une image A3 d'une foule dense et une image d'un personnage (Charlie), prouvez que vous savez où il se trouve sans divulguer sa position.

b. Étant donné un graphe dessiné sur une feuille A3, prouvez que vous savez le colorier avec 3 couleurs sans rien divulguer du coloriage.

Exercice 2 ★★ (Avec machine) Cet exercice utilise la bibliothèque `PyNaCl` de Python pour pratiquer l'interface d'un système de signature numérique. Cette bibliothèque fournit une interface Python à la bibliothèque C++ NaCl. Cette bibliothèque implémente notamment le système Ed25519 de signature à clef publique, dont l'interface correspond à ce que l'on a décrit en 3.2.3. La documentation officielle est disponible à

`https://pynacl.readthedocs.io/en/latest/signing/`

a. Importez le module `nacl.signing` et utilisez la fonction `SigningKey` pour générer une clef de signature nommée `maclef`.

b. Récupérez les clefs publique de `maclef` dans une variable `pub` et affichez la sous la forme d'un mot hexadécimal.

c. Déclarez une chaîne de caractères `message`, initialisez la comme bon vous semble, puis calculez la signature `masig` de `message` par `maclef`. Affichez `masig` sous la forme d'un mot hexadécimal.

d. Créez la clef de vérification associée à `maclef` et `verif`. Créez une nouvelle chaîne de caractères `nouveau_message` et initialisez le avec le même texte que celui que vous avez choisi pour `message`. Vérifiez au moyen de `verif` que `masig` est bien accepté comme la signature de `nouveau_message` par `maclef`.

e. Changez le contenu de `message` et vérifiez au moyen de `verif` que `masig` n'est pas accepté comme la signature de `message` par `maclef`.

Exercice 3 ★★ (Optionnel, avec machine) Récupérez sur la page arche du cours l'archive `exercice_signature.zip`. Elle contient une clef publique, une signature et cinq messages. Déterminez si la signature correspond à la signature par cette clef publique d'un des messages, et si oui lequel.

3.3 Signature de Schnorr

Plusieurs systèmes de signature numérique sont actuellement utilisés. S'ils varient sur un certain nombre d'aspects techniques (importants !), les principes de ces systèmes sont assez similaires. Nous examinons tout d'abord le protocole de signature de Schnorr et esquisserons ensuite deux de ses variantes (DSA et ECDSA).

3.3.1 Logarithme discret

Les systèmes de signature que l'on va voir mettent en jeu quelques notions d'arithmétique élémentaire. On renvoie à l'annexe A pour quelques rappels de base (supposés ici connus).

Fixons un groupe multiplicatif fini $(G, \cdot)$ d'élément neutre 1. Pour tout $g \in G \setminus \{1\}$ et $n \in \mathbb{Z}$, on note

$$g^n = \underbrace{g \cdot g \cdot \ldots \cdot g}_{n \text{ fois}}$$

avec la convention que $g^0 = 1$. Si $y = g^x$, on dit que y est l'*exponentielle* de x (de base g, dans G) et x est *un logarithme discret* de y (de base g, dans G).

Par exemple, dans $G \stackrel{\text{def}}{=} \mathbb{Z}/7\mathbb{Z}$ on a $2^5 = 4$. Ainsi, 4 est l'exponentielle de 5 (et 5 un logarithme de 4) de base 2 dans $\mathbb{Z}/7\mathbb{Z}$.

3.3.2 Signature de Schnorr

Le système de signature de Schnorr fixe quatre paramètres (q, H, G, g) connus de tous les utilisateurs : un entier q premier, une fonction de hachage H à valeurs dans $\mathbb{Z}/q\mathbb{Z}$, un groupe G d'ordre[9] q, et un élément $g \in G$ qui engendre G. Le groupe G est choisi de sorte qu'il soit facile d'y calculer l'exponentielle mais difficile d'y calculer le logarithme.

La fonction `genere_clef` choisit un entier c_{sec} aléatoire uniformément dans $\{1, 2, \ldots, q-1\}$ et calcule $c_{pub} \stackrel{\text{def}}{=} g^{c_{sec}}$ dans G. La paire (c_{pub}, c_{sec}) est la clef, c_{pub} étant la *clé publique* et c_{sec} étant la *clé secrète*. Comme c_{sec} est un logarithme discret de c_{pub}, calculer la clef secrète à partir de la clef publique exige de résoudre une instance d'un problème considéré comme difficile.

9. Soulignons que pour q premier, il n'existe qu'un seul groupe d'ordre q *à isomorphisme près*. Un tel isomorphisme peut cependant être difficile à expliciter et le choix d'un groupe particulier peut influer sur la difficulté du calcul du logarithme discret.

Pour *signer* un mot binaire m par c_{sec}, on commence par choisir un entier k aléatoire uniformément dans $\{1, 2, \ldots, q-1\}$. On calcule ensuite $r \stackrel{\text{def}}{=} g^k$, $e \stackrel{\text{def}}{=} H(\text{bin}(r) \cdot m)$ et $s \stackrel{\text{def}}{=} k - c_{sec}e$. La signature est la paire (s, e).

Pour *vérifier* que (s, e) est une signature valide d'un mot binaire m par une clef de partie publique c_{pub}, on calcule $r_v \stackrel{\text{def}}{=} g^s(c_{pub})^e$ puis $e_v \stackrel{\text{def}}{=} H(\text{bin}(r_v) \cdot m)$. On accepte la signature comme valide si et seulement si $e_v = e$.

3.3.3 Système d'identité numérique

Pour construire un système d'identité numérique il suffit de rendre public le choix d'un système de signature et ses éventuels paramètres (les valeurs choisies pour q, H, G, et g dans le cas du système de Schnorr). Dans un tel système :

- N'importe qui peut créer une clef (c_{pub}, c_{sec}).
- Toute clef publique c_{pub} peut servir d'identité publique.
- Lorsque c_{pub} diffuse un contenu numérique (déclaration, flux vidéo, fichier de données, ...), elle peut signer ce contenu. Toute personne pourra vérifier que la signature est correcte, et ainsi [10] que le contenu émane de quelqu'un connaissant la clef secrète associée à c_{pub}.

Ce système est *décentralisé* car il n'y a pas besoin de stocker en un quelconque endroit la liste des identifiants existants. Tout utilisateur connaissant les paramètres du système de signature utilisé peut créer et gérer ses identifiants en toute autonomie.

3.3.4 Exercices

Exercice 4 ★ Examinons la signature de Schnorr pour les paramètres $q = 5$, $G = \mathbb{Z}_5$, $g = 2$ et

$$H : \left\{ \begin{array}{rcl} \{0,1\}^* & \to & \mathbb{Z}_5 \\ w & \mapsto & \text{bin}(w) \mod 5 \end{array} \right.$$

a. Donnez un exemple de paire (y, x) de clef, y étant publique et x étant privée.

b. Donnez toutes les paires de clefs appariées possibles.

c. Calculez la signature du mot binaire $m \stackrel{\text{def}}{=} 110110$ pour la clef publique $y = 4$ lorsque l'entier aléatoire vaut $k = 3$.

d. Vérifiez que cette signature est bien acceptée par le système de Schnorr.

Exercice 5 ★★ Justifiez que dans le protocole de Schnorr, une signature valide est toujours acceptée.

Exercice 6 ★★ Supposons que l'on dispose de m, y, s et e, où m est un mot binaire, y une clef publique et (s, e) la signature de m par une clef de clef publique y selon le protocole de Schnorr.

a. Supposons aussi que $m' \neq m$ soit un mot binaire tel que (s, e) est aussi accepté comme signature de m' par une clef de clef publique y. Peut-on construire une collision pour H ?

b. Supposons que l'on arrive à déterminer l'entier k aléatoire utilisé pour signer m. Que peut-on en déduire ?

10. Ça c'est la théorie. En pratique, cette implication est à examiner soigneusement : elle n'est valide que sous l'hypothèse (raisonnable) que la primitive cryptographique n'a pas été cassée et (moins évident) que les protocoles de publication des clefs, de diffusion des messages, d'exécution des codes de vérification, ...n'ouvrent pas des failles de sécurité.

3.4 Signatures DSA et ECDSA

Le protocole de signature de Schnorr étant breveté, d'autres systèmes de signature similaires ont été développés dans le but d'offrir des alternatives librement utilisables. Ces variantes sont un peu plus compliquées que le système de Schnorr, mais ont néanmoins été plus largement déployées. [11] On présente ici rapidement deux de ces variantes : DSA et ECDSA.

3.4.1 Le système DSA

Un premier paramètre du système DSA est la *taille de clef*, que l'on note ici ℓ. Un choix courant est $\ell = 2048$ mais pour des clefs dont on souhaite qu'elles restent sûres au-delà de 2030, le NIST recommande de prendre $\ell = 3072$.

Le système de signature DSA fixe quatre paramètres (p, q, H, g) connus de tous les utilisateurs. Il y a tout d'abord deux entiers premiers p et q tels que p est sur ℓ bits, q est sur 256 bits et q divise $p-1$. Ensuite vient une fonction de hachage H de taille 256 bits (typiquement SHA-256). Enfin, g est un générateur du groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z})^*$ dans lequel on travaille, calculé comme suit : on choisit un entier aléatoire h uniformément au hasard dans $\{2, 3, \ldots, p-2\}$, et on calcule $g \stackrel{\text{def}}{=} h^{\frac{p-1}{q}} \mod p$; dans l'éventualité (peu probable, mais possible) où $g = 1$, on répète l'opération.

La fonction `genere_clef` choisit un entier x aléatoire uniformément dans $\{1, 2, \ldots, q-1\}$ et calcule $y \stackrel{\text{def}}{=} g^x \mod p$. La paire (y, x) est la clef, x étant secret et y étant public. Comme x est le logarithme discret de y, calculer la clef secrète à partir de la clef publique exige de résoudre une instance d'un problème considéré comme difficile.

Pour *signer* un mot binaire m par x, on commence par choisir un entier k aléatoire uniformément dans $\{1, 2, \ldots, q-1\}$ et on calcule deux nombres :

$$r \stackrel{\text{def}}{=} (g^k \mod p) \mod q \qquad \text{et} \qquad s \stackrel{\text{def}}{=} \left(k^{-1}\left(H(m) + xr\right)\right) \mod q.$$

La *signature* de m par x est la paire (r, s).

Pour *vérifier* que (r, s) est une signature valide d'un mot binaire m par une clef de partie publique y dans un système de paramètres (p, q, g), on procède aux vérifications suivantes :

- on doit avoir $0 < r < q$ et $0 < s < q$,
- on calcule

$$u_0 \stackrel{\text{def}}{=} s^{-1} \mod q, \qquad u_1 \stackrel{\text{def}}{=} H(m) \cdot u_0 \mod q, \qquad u_2 \stackrel{\text{def}}{=} r \cdot w \mod q.$$

- on calcule $v \stackrel{\text{def}}{=} (g^{u_1} \cdot y^{u_2}) \mod q$ et on accepte la signature si $v = r$.

3.4.2 Groupe associé à une courbe elliptique et ECDSA

Une *courbe algébrique* est un ensemble de points défini comme le lieu d'annulation d'un polynôme. Le cercle d'équation $X^2 + Y^2 = 2$ dans le plan affine $\mathbb{R}^2$ est un exemple familier, mais ce polynôme définit aussi une courbe algébrique sur d'autre corps. Ainsi,

$$\{(1,1), (1,2), (2,1), (2,2)\}$$

est l'ensemble des points de $\mathbb{Z}/3\mathbb{Z} \times \mathbb{Z}/3\mathbb{Z}$ qui annulent ce polynôme ; c'est aussi une courbe algébrique. Plus généralement, on peut s'intéresser au lieu d'annulation d'un polynôme k-varié sur $\mathbb{F}^k$, où $\mathbb{F}$ est un corps fini.

Le système de signature utilisé par BITCOIN repose sur une courbe précise. Notons $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. Il s'avère que p est premier, aussi $\mathbb{F}_p \stackrel{\text{def}}{=} \mathbb{Z}/p\mathbb{Z}$ est un corps. L'ensemble des points de $\mathbb{F}_p \times \mathbb{F}_p$ satisfaisant

$$Y^2 = X^3 + 7 \tag{3.1}$$

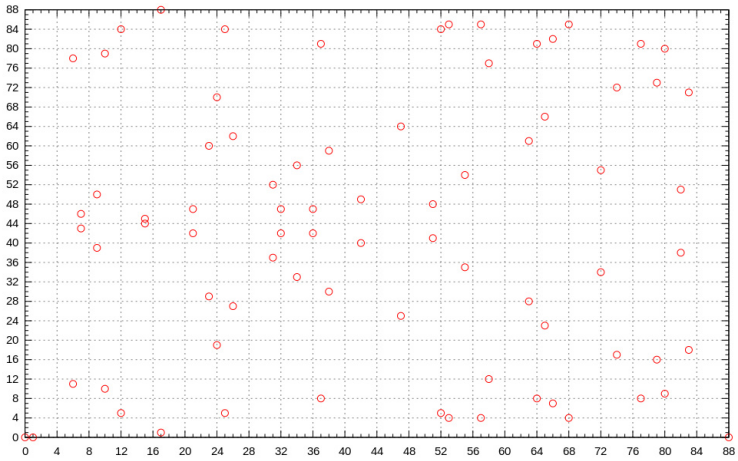


FIGURE 3.1 – Exemple de courbe elliptique : $Y^2 = X^3 - X$, sur le corps $\mathbb{Z}/89\mathbb{Z}$. (Source : Wikipédia.)

est une courbe appelée `secp256k1`. Comme (presque) toute équation de la forme $Y^2 = X^3 + aX + b$ où a et b sont des constantes, c'est une *courbe elliptique*. Cette courbe est, comme l'exemple représenté en figure 3.1, un ensemble fini de points.

On définit une *droite* sur $\mathbb{F} \times \mathbb{F}$, où $\mathbb{F}$ est un corps fini, comme le lieu d'annulation d'un polynôme multivarié de degré 1. Les axiomes d'Euclide gouvernent cette géométrie comme le monde affine ; en particulier, par deux points d'un corps fini passent une et une seule droite, et deux droites non parallèles se coupent en un et un seul point. Les courbes elliptiques (sur des corps finis) ont deux propriété remarquables :

a. une droite qui contient au moins deux points de la courbe en contient exactement trois, et

b. la courbe admet une symmétrie par rapport à l'axe des X.

On peut se servir de ces propriétés pour définir une *loi de groupe* sur la courbe : étant donné deux points P et Q de la courbe, on définit $P \odot Q$ comme le symmétrique, relativement à l'axe des X, du troisième point de la courbe sur la droite PQ.

L'algorithme ECDSA réalise une signature cryptographique similaire à DSA, mais elle s'appuie pour cela sur l'opération de logarithme discret dans le groupe défini par une courbe elliptique. Faute de temps, on ne détaille pas cet algorithme ici et on renvoie à :

`https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm`

3.5 Exercices supplémentaires

Exercice 7 ★ *(remake pour DSA de l'exercice 5b).* Une personne de clef publique y nous transmet la signature (r, s) d'un message m dans le système DSA. En testant quelques générateurs pseudo-aléatoires mal initialisés, on a réussit à déterminer l'entier aléatoire k utilisé lors de cette signature. Que peut-on faire de cette information ?

Exercice 8 ★ **(Examen d'un document technique)** L'agence nationale (Française) de la sécurité des systèmes d'informations (ANSSI) fait des préconisations en matière de cryptographie dans deux guides :

`https://www.ssi.gouv.fr/guide/mecanismes-cryptographiques/`

a. Parmi les trois systèmes de signature numérique que l'on a présenté (Schnorr, DSA et ECDSA), lequel ou lesques sont approuvés par l'ANSSI ?

11. Le brevet aux USA sur le protocole de Schnorr n'a expiré qu'en 2008.

b. Quelle est la condition de cette approbation ?

c. Lequel des exercices précédents aborde le sujet de la note 6.2b du *Guide de sélection d'algorithmes cryptographiques* ?

Exercice 9 ★★ (Logarithme discret) Soit G un groupe multiplicatif fini, g un générateur de G et ℓ l'ordre de g, c'est à dire le plus petit entier tel que $g^\ell = 1$. Montrer que pour tout $y \in G$ il existe un entier k tel que

$$n \text{ est un logarithme de } y \text{ de base } g \quad \Leftrightarrow \quad n = k \mod \ell.$$

Notes personnelles

Chapitre 4

Décentraliser la tenue du registre par élections et consensus

Les séances 1 à 3 ont posé les notions de cryptographie permettant de construire un registre inaltérable d'enregistrement dont les auteurs sont authentifiables via leurs signatures. Cette séance amorce la décentralisation de la tenue de ce registre. Comme le registre est souhaité inaltérable, cette tenue consiste à l'ajout de nouveaux blocs. Elle est *décentralisée* si elle est réalisée conjointement par une communauté d'agents sans qu'aucun d'entre eux ne dirige particulièrement les opérations. Une approche classique ajoute chaque bloc en deux étapes :

- On organise une *élection* pour désigner un des agents de la communauté, afin que celui-ci rédige une proposition de nouveau bloc.
- On constitue un *consensus* entre les agents de la communauté afin d'accepter ou de rejeter ce bloc pour toutes les copies du registre.

Ainsi, chaque ajout d'un nouveau bloc suscite l'organisation d'une élection et la constitution d'un consensus. Dans ce chapitre, nous examinons comment ces problématiques sont formalisées par le *calcul distribué* et traitées par l'*algorithmique distribuée*.

Les objectifs sont que vous sachiez...

- identifier les caractéristiques d'un modèle de calcul distribué,
- formuler des problèmes algorithmiques en calcul distribué,
- concevoir des algorithmes distribués élémentaires,
- analyser des algorithmes distribués simples.

4.1 Modèles et problèmes

Informellement, un système distribué est un ensemble de machines, chacune capable de calculer, qui sont en mesure de communiquer entre elles. En première approximation, on souhaite faire résoudre un problème à un système distribué en installant sur chaque machine *le même* programme, puis en exécutant ce programme sur *toutes* les machines *à la fois*. La formalisation que l'on présente ici est dite « à réseau » ; soulignons que ce n'est pas la seule possible, un autre modèle classique étant « à mémoire partagée ».

4.1.1 Machines, réseau et identifiants

Dans ce cours, un **système distribué** est un ensemble fini de machines, reliées entre elles par un réseau de communication.

Chaque machine est un système de calcul classique, que l'on modélise ici par le modèle RAM. [1] Certaines paires de machines sont connectées par des canaux de communication qui permettent d'envoyer et de recevoir des messages. On modélise ce réseau par un graphe $G = (V, E)$ dont les sommets V sont

1. Ce pourrait tout aussi bien être un automate fini, une machine de Turing, ...

les machines et les arêtes E sont les paires de machines connectées. Chaque machine peut déclencher l'envoi d'un ou plusieurs messages, peut recevoir des messages et peut réagir aux messages reçus. Un même programme peut donc s'exécuter différemment sur chaque machine en réagissant aux messages qu'elle reçoit.

Les machines connectées à une machine u sont appelées les *voisines* de u. Le nombre de voisines de u est le *degré* de u, noté $\deg u$. Chaque machine u connaît son nombre $\deg u$ de voisines, et sait les distinguer entre elles. Chaque machine u dispose de canaux d'émission (numérotés de 1 à $\deg u$) et de canaux de réception (numérotés de 1 à $\deg u$). Ces numérotations sont cohérentes entre émission et réception sur une même machine, mais ne sont pas a priori cohérentes d'une machine à l'autre. [2] En particulier, chaque machine ne connaît le réseau que *localement*.

On suppose ici que les machines d'un système distribué sont identiques à l'exception d'un **identifiant unique**, modélisé par une variable interne noté `id` et initialisée différemment sur chaque machine.

4.1.2 Présentation des algorithmes

La description des algorithmes suit les règles usuelles. Il est pratique de normaliser les instructions de communication sur le réseau :

`emettre(I,m)` prend en entrée un ensemble I d'entiers et un mot binaire m, et envoie m sur les canaux d'index dans I. On s'autorise à écrire, pour un entier i, `emet(i,m)` pour `emet({i},m)` et `emet(tous,m)` pour émettre sur *tous* les canaux.

`recevoir` collecte les messages transmis par les voisins.

Une machine est *active* au début du calcul et devient *inactive* lorsqu'elle a terminé l'exécution du programme. Une machine inactive ne peut pas redevenir active. [3] Pour plus de clarté dans la présentation des algorithmes, on matérialise la fin d'exécution d'un programme par les instructions suivantes.

`choisir(r)` indique la valeur r choisie comme résultat du calcul pour cette machine

`terminer` conclut l'exécution du programme sur cette machine.

4.1.3 Retards, pannes et dysfonctionnements

L'étude des algorithmes distribués attache une grande importance à leur capacité à supporter certains imprévus.

Une machine dans un système distribué a un *comportement Byzantin* si elle ne se comporte pas conformément à l'algorithme. Cela peut être dû à un problème technique (défaillance d'un des ports de communication par exemple) ou à un comportement malicieux (prise de contrôle de la machine et modification du programme par un tiers malveillant). Lorsque l'on modélise un système où des comportements Byzantins sont possibles, on dit qu'une machine est *fiable* si elle se comporte conformément à l'algorithme.

Dans un système distribué, une *panne* désigne le fait qu'une des machines du système *arrête* de fonctionner : elle ne reçoit plus les messages, n'effectue plus aucun calcul interne et n'émet plus de messages. Une panne est un problème moins général qu'un comportement Byzantin mais suffit déjà à modéliser la possibilité de plantage, de coupure d'électricité, d'incendie, ... Une machine qui n'est pas en panne est appelée *fonctionnelle*.

2. Autrement dit, si les messages écrits par une machine A écrit sur son canal #1 sont reçus par une machine B, alors les messages que A reçoit sur son canal #1 viennent aussi de B. En revanche, le canal de B qui le connecte à A n'est pas nécessairement numéroté #1 pour B.

3. On traitera séparément les questions de redémarrage après panne en Section 4.4.

4.1.4 Problèmes d'élection et de consensus

Dans leur forme la plus simple, les problèmes qui nous intéressent se formalisent comme suit :

Élection (sans panne)

Entrée : Rien

Sortie : Chaque machine choisit 0 ou 1. Exactement une machine choisit 1.

Consensus (sans panne)

Entrée : Chaque machine a un vote initial valant 0 ou 1.

Sortie : Chaque machine choisit un vote final valant 0 ou 1. Ces votes doivent satisfaire les conditions suivantes :

a. Toutes les machines ont le même vote final.

b. Le vote final égale l'un des votes initiaux.

En l'absence de panne, ces deux problèmes admettent des approches simples. Pour l'élection, il suffit que les machines s'échangent leurs identifiants respectifs, puis appliquent toutes une même règle pour déterminer l'identifiant élu (par exemple, l'élue est la machine de plus grand identifiant). Pour le consensus, il suffit que les machines s'échangent leurs votes initiaux, puis appliquent toutes une même règle pour déterminer le vote final (par exemple voter 1 si tout le monde a ce vote initial et 0 sinon).

Pour mettre en œuvre ces approche, il convient d'organiser le partage d'information. Nous allons voir que les modalités précises de ce partage peuvent dépendre de la topologie du réseau ou de la possibilité pour les machines de se synchroniser.

4.2 Modèle synchrone sans panne

Notre premier modèle de calcul est dit **synchrone** car les machines commencent à exécuter leurs programmes en même temps, et peuvent s'attendre pour rester synchronisées. Pour la présentation d'algorithmes dans les modèles synchrones, on pourra utiliser l'instruction :

`synchro` attend que l'ensemble des machines du système distribué atteignent ce point de l'algorithme avant de continuer.

On appelle *phase* un intervalle de temps entre deux synchronisations successives.

On commence par présenter des algorithmes d'élection et de consensus élémentaires mais limités à des topologies de réseau particulières. On analyse ensuite, en exercice, un algorithme valide pour tout réseau (connexe).

4.2.1 Cas d'un réseau complet

Quand le réseau est un graphe complet, c'est à dire que toute paire de sommets forme une arête, l'échange d'information peut se faire en une phase :

4.2.2 Cas d'un réseau en anneau

Considérons maintenant le cas où le réseau forme un anneau. On suppose que l'anneau est « orienté » : chaque machine a un voisin nommé `gauche` et un voisin nommé `droite`, et chaque machine est le voisin `gauche` de son voisin `droite` (et réciproquement). L'algorithme suivant, dû à Lelann-Chang-Robert, donne une première solution pour l'élection (on en verra d'autres en exercices).

```
emettre(tous, id)
synchro
recevoir
m = id max reçu
si id == m
   choisir(1)
sinon
   choisir(0)
terminer
```

```
emettre(tous, vote initial)
synchro
recevoir
v = min(vote initial, votes recus)
choisir(v)
terminer
```

FIGURE 4.1 – Algorithme d'élection (à gauche) et de consensus (à droite) pour un réseau complet.

```
1  record = id
2  emettre(droite,id)
3  synchro
4  recevoir
5  si reçu un identifiant i>record:
6      record = i
7      emettre(droite,record)
8  si reçu un identifiant i == id:
9      emettre(droite,''STOP'')
10     choisir(1)
11     terminer
12 si reçu ``STOP''
13     emettre(droite,''STOP'')
14     choisir(0)
15     terminer
16 retourner à la ligne 3
```

En clair, les machines font circuler les identifiants le long du réseau. Une machine ne propage un identifiant que s'il est le plus grand vu qu'elle a vu jusqu'alors, le sien compris. Ainsi, la machine M de plus grand identifiant bloque tous les messages, et son identifiant est le seul à faire « le tour du réseau ». Cette machine M finira par recevoir son propre identifiant, se considère élue et informe les autres, qui se considèrent alors non-élues.

On peut facilement adapter cet algorithme pour résoudre le consensus, par exemple en choisissant le vote initial de la machine élue et en propageant ce vote avec les messages « STOP ».

4.2.3 Exercices

Exercice 1 ★★ Dans un réseau torique à $n \times m$ machines, les machines sont numérotées par les paires $(i, j) \in [n] \times [m]$. La machine (i, j) a pour voisins $(i, j+1)$ (« haut »), $(i, j+1)$ (« bas »), $(i-1, j)$ (« gauche ») et $(i+1, j)$ (« droite »). Le premier indice est considéré modulo n et le second modulo m.

Adaptez l'algorithme de Lelann-Chang-Robert pour un système distribué à réseau torique dont le nombre de machines est inconnu, et où chaque machine dispose d'un identifiant unique.

Exercice 2 ★★ Intéressons-nous à l'estimation asymptotique de l'efficacité de l'algorithme de Lelann-Chang-Robert sur un réseau en anneau de n machines.

a. Quel est le nombre de messages échangés dans le pire cas ?

b. Combien de phases dure l'exécution de l'algorithme dans le pire cas ?

Signalons qu'il existe des solutions de meilleure *complexité de communication* que l'algorithme de Lelann-Chang-Robert, par exemple l'*algorithme de Hirschberg-Sinclair*.

Exercice 3 ★★★ L'algorithme ci-dessous résout le problème de l'élection sur un réseau quelconque dès lors qu'il est connexe. Les opérations $-$ et $+$ des lignes 6 et 9 sont, respectivement, la différence et l'union entre ensembles.

a. Vérifiez que cet algorithme fonctionne correctement pour un réseau complet et un réseau en anneau. En combien de phase termine-t-il dans chacun des cas ? (On considère que l'algorithme termine lorsque la dernière machine termine.)

b. Supposons maintenant le réseau arbitraire (mais toujours connexe). Remarquons qu'à chaque phase, l'algorithme exécute une instruction d'émission (ligne 2 ou ligne 8). Choisissons une machine a.

   b1. Décrivez l'ensemble des sommets qui reçoivent l'identifiant de a lors de la $i$ème instruction d'émission. *(On pourra raisonner par récurrence.)*

   b2. Réciproquement, lors de la $i$ème phase, quels sont les sommets dont REC contient l'identifiant après exécution de la soustraction ligne 6 ?

   b3. À quelle phase la machine a exécute-t-elle les lignes 12-16 ?

c. En combien de phases cet algorithme termine-t-il dans le pire cas ?

```
1  S = {id}
2  emettre(tous, S)
3  synchro
4  recevoir
5  REC = ensemble des identifiants nouvellement reçus
6  NOUV = REC - S
7  si NOUV est non-vide:
8      transmettre(tous, NOUV)
9      S = S + NOUV
10     retourner ligne 3
11 sinon:
12     si max de S == id:
13         choisir(1)
14     sinon
15         choisir(0)
16     terminer
```

4.3 Modèle asynchrone sans panne

Notre second modèle est **asynchrone**, c'est à dire qu'on n'y fait aucune hypothèse sur les vitesses relatives d'exécution des programmes sur les différentes machines, ni sur le temps d'acheminement des messages. Plus précisément :

- on ne dispose plus de l'instruction `synchro` permettant de synchroniser les machines,
- le temps que prend la transmission d'un message est *fini*, mais n'est pas borné *a priori*,
- le temps que prend l'exécution d'une instruction sur un ordinateur est *fini*, mais n'est pas borné *a priori*,

Dans ce cours, nous supposerons de plus que chaque canal de communication se comporte comme une file (FIFO), c'est à dire que les messages sortent *de ce canal* dans l'ordre dans lequel ils sont entrés *dans ce canal*.[4]

Une **exécution** d'un algorithme asynchrone sur un système asynchrone est une séquence d'événements élémentaires, chacun étant de l'un des deux types suivants :

4. Cela n'a rien d'évident : si le canal de communication est réalisé par un réeau offrant plusieurs possibilités de routage, certaines routes peuvent être plus rapides que d'autres.

- *une* des machines exécute l'instruction suivante dans son programme,
- *un* des messages en transit sur *un* des canaux arrive à destination.

Une exécution **a terminé** lorsque toutes les machines ont terminé l'exécution de leur programme et tous les canaux sont vides. Un algorithme asynchrone **résout** un problème si **toutes** les exécutions possibles de cet algorithme terminent et que tous les états finaux du système résolvent ce problème.

On limite dans cette section la discussion au problème de l'élection. Les arguments utilisés s'adaptent facilement au problème du consensus.

4.3.1 Un premier exemple simple

Commençons par examiner un algorithme élémentaire d'élection dans un système asynchrone à réseau complet :

```
1 emettre(tous, id)
2 attendre d'avoir reçu un message de chaque voisin
3 calculer m = id max reçu
4 si id == m
5    choisir(1)
6 sinon
7    choisir(0)
8 terminer
```

On a simplement repris l'algorithme de la Section 4.2.1 pour l'élection dans un système synchrone à réseau complet, et on a remplacé, ligne 2, l'instruction `synchro` par une instruction d'attente d'avoir reçu un message de chaque voisin.

Il peut sembler "évident" à première vue que toutes les exécutions possibles de cet algorithme résolvent l'élection. On peut prouver cela en explicitant l'ensemble des informations qui décrivent l'**état du système** à un instant donné, puis en établissant des propriétés vérifiées par cet état.

Dans l'exemple ci-dessus, l'état du système peut consister en :

- pour chaque machine, la prochaine ligne du programme à exécuter,
- pour chaque machine, la liste des identifiants d'autres machines qu'elle connait,
- pour chaque canal de communication, la liste ordonnée des messages en transit.

Une propriété est dite de **vivacité** (*liveness*) si (i) dans toute exécution de l'algorithme, il existe au moins un état du système qui la satisfait, et (ii) une fois qu'un état satisfait la propriété, tout état ultérieur la satisfait aussi. En voici quelques exemples pour l'algorithme ci-dessus :

> Chaque machine connaît l'identifiant de toutes les autres machines.
> Chaque machine a fait un choix dans $\{0, 1\}$.
> Chaque machine termine.

Une propriété est dite de **sûreté** (*safety*) si dans toute exécution de l'algorithme, tout état du système satisfait cette propriété. En voici quelques exemples pour l'algorithme ci-dessus :

> Aucune machine d'identifiant maximal n'a choisi 0.
> Aucune machine d'identifiant non maximal n'a choisi 1.

Chaque propriété doit être établie formellement. Illustrons cela :

Lemme 5. *« Chaque machine connaît l'identifiant de toutes les autres machines » est une propriété de vivacité.*

Démonstration. Pour $1 \leq i \leq n$ notons t_i le temps pris par l'exécution de la première instruction du programme de la $i$ème machine. Cette exécution occasionne, pour tout $1 \leq j \leq n$, $j \neq i$, l'envoi d'un message destiné à la $j$ème machine et contenant l'identifiant de la $i$ème machine ; notons $t_{i,j}$ le temps mis par ce message pour arriver à destination. Dans le modèle de calcul asynchrone, dans toute exécution les valeurs t_i et $t_{i,j}$ sont finies. Ainsi, dans toute exécution il existe un temps $T = (\max_i t_i) + (\max_{i,j} t_{i,j})$ tel que dans tout état atteint après le temps T, chaque machine j a reçu de chaque machine $i \neq j$ un message contenant l'identifiant de i. □

Chacune des six propriétés ci-dessus peut être prouvée de la même manière. On peut ensuite les combiner pour prouver que l'algorithme résout bien l'élection puisque dans toute exécution :

- chaque machine termine l'exécution de son programme après avoir choisi 0 ou 1,
- la machine d'identifiant maximal a choisi 1,
- les autres machines ont choisi 0.

4.3.2 Un exemple plus compliqué

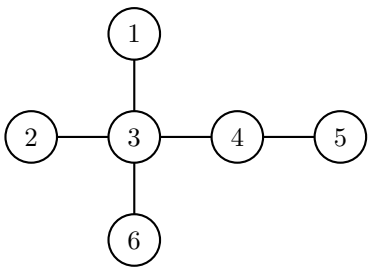
Nous venons de constater qu'un algorithme que l'on avait formulé pour le modèle synchrone n'avait trivialement pas besoin de synchronisation. Qu'en est-il pour l'algorithme étudié à l'exercice 3 ci-dessus ? Il est naturel de remplacer la synchronisation par...

```
1  S = {id}
2  emettre(tous, S)
3  attendre tant qu'aucun nouveau message n'est arrivé
4  REC = ensemble des identifiants nouvellement reçus
5  NOUV = REC - S
6  si NOUV est non-vide:
7      transmettre(tous, NOUV)
8      S = S + NOUV
9      retourner ligne 3
10 sinon:
11     si max de S == id:
12         choisir(1)
13     sinon
14         choisir(0)
15     terminer
```

Nous allons examiner en exercice si cet algorithme résout l'élection.

4.3.3 Exercices

On va examiner des exécutions possibles de l'algorithme de la section 4.3.2 sur le réseau suivant :



Dans ce qui suit, on indexe chaque variable par la machine considérée (ainsi $\texttt{S}_5$ est la variable `S` de la machine 5).

Exercice 4 ★ Commençons par examiner quelques exécutions :

a. Considérons une exécution qui commence par 2 pas de calcul de 4, puis l'arrivée du premier message $4 \to 5$, puis 5 pas de calcul de 5, puis l'arrivée du premier message $5 \to 4$, puis 2 pas de calcul de 4. À ce stade, que valent $\texttt{REC}_4$ et $\texttt{REC}_5$?

b. Même question, mais cette fois-ci l'exécution commence par 4 pas de calcul de 4, puis 5 pas de calcul de 5, puis l'arrivée du premier message $4 \to 5$, puis l'arrivée du premier message $5 \to 4$.

Exercice 5 ★★ Intéressons-nous maintenant à des propriétés :

a. Prouvez que « $\texttt{S}_6$ contient 3 » est une propriété de vivacité.

b. « $\texttt{S}_6$ contient 1 » est-elle une propriété de vivacité ?

c. Prouvez que « $\texttt{S}_6$ induit un sous-graphe connexe » est une propriété de sûreté.

Exercice 6 ★★★ Prouvez que l'algorithme ne résout pas le problème de l'élection.

4.4 Modèle synchrone avec pannes non-Byzantines

Pour clore ce chapitre, examinons le problème du consensus en présence de pannes, que l'on suppose ici non-Byzantines.

4.4.1 Redéfinition du problème

On revient au modèle synchrone et on suppose le réseau complet et les machines à identifiants uniques. Dans ce cadre, le problème de consensus doit être reformulé ainsi :

Consensus tolérant aux pannes

Entrée : Chaque machine a un vote initial valant 0 ou 1.

Sortie : Chaque machine choisit un vote final valant 0 ou 1. Ces votes doivent satisfaire les conditions de cohérence globale suivantes :

a. Toutes les machines fonctionnelles ont le même vote final.

b. Le vote final égale l'un des votes initiaux.

Notons bien que l'on ne demande pas à ce que le vote final égale le vote initiale d'une machine fonctionnelle.

Un algorithme distribué *résout le consensus en tolérant k pannes* si toute exécution de cet algorithme au cours de laquelle au plus k machines tombent en panne aboutit, pour les machines restant fonctionnelles, à une solution. Soulignons que chaque machine qui tombe en panne peut le faire à *n'importe quel* moment lors de l'exécution de son programme.

4.4.2 Cernons les difficultés

Commençons par clarifier que l'algorithme « d'unanimité » donné en section 4.2.1 ne résout *pas* le consensus en tolérant ne serait-ce qu'une panne. Pour voir cela, il est utile de le reformuler comme suit :

```
1 pour j = 1 .. nombre de voisins
2    emettre(j, vote initial)
3 synchro
4 recevoir
5 v = min(vote initial, votes recus)
6 choisir(v)
7 terminer
```

Considérons un système à 3 machines de votes initiaux 1, 1 et 0. Supposons que la machine de vote initial 0 tombe en panne après avoir émis son vote initial *exactement une fois*. Sur les deux machines fonctionnelles à la fin de l'algorithme, l'une choisira $\min(0, 1, 1)$ et l'autre $\min(1, 1)$.

4.4.3 Une solution par rediffusion

Considérons l'algorithme suivant, dont certaines opérations sont clarifiées après l'algorithme.

```
1  V[0] = {}
2  V[1] = {(id, vote initial)}
3  pour j = 1 .. f+1:
4     N =  V[j] - V[j-1]
5     pour k = 1 .. nombre de voisins
6        emettre(k,N)
7     synchro
8     recevoir
9     V[j+1] = V[j]
10    pour chaque V reçu:
11       V[j+1] = V[j+1] union V
12 vote = calcul(V[f+2])
13 choisir(vote)
```

L'entier `f` est un paramètre fixé avant le déploiement de l'algorithme. On considère chaque `V[j]` comme un ensemble (sans répétition, donc) et l'opération "-" de la ligne 4 correspond à la différence entre les deux ensembles. La fonction `calcul` de la ligne 12 applique n'importe quelle règle déterministe qui choisit une valeur parmi celles de l'ensemble fourni en argument (par unanimité, majorité, ...). Nous allons établir en exercice que cet algorithme résout le consensus distribué en tolérant `f` pannes.

4.4.4 Exercices

Exercice 7 ★ Revenons sur l'algorithme « unanimité » discuté en section 4.4.2.

a. Décrire une panne qui le fait échouer dans un système de 1001 machines.

b. Est-ce possible de faire en sorte que la panne fasse que la moitié des machines fonctionnelles vote 0 et l'autre moitié vote 1 ?

c. Même question qu'au (b) si on remplace la ligne 5 par une règle de majorité.

d. Même question qu'au (b) si chaque machine émet son vote initial *et* son identifiant, et que l'on choisisse le vote de la machine d'identifiant maximal.

Exercice 8 ★ Examinons le fonctionnement de l'algorithme de la section 4.4.3 pour 3 machines d'identifiants 1, 2 et 3 et de votes initiaux 0, 1 et 0, respectivement.

a. Supposons que l'algorithme soit exécuté sans panne avec `f`$= 0$. Que vaut $V[2]$ sur chacune de ces machines ? Est-ce qu'un consensus est trouvé quel que soit `calcul` ?

b. Supposons maintenant que l'algorithme soit exécuté avec `f`$= 0$ et que la machine 3 tombe en panne à la ligne 6 pour $j = 1$ entre les valeurs de $k = 1$ et $k = 2$ (c'est-à-dire qu'elle transmet son N uniquement à la machine 1). Que vaut $V[2]$ pour les machines 1 et 2 ? Proposez une règle de `calcul` pour laquelle le consensus échoue.

c. Supposons maintenant que l'algorithme soit exécuté avec `f`$= 1$ et que la machine 3 tombe en panne au même point qu'à la question (b). Que vaut $V[3]$ pour les machines 1 et 2 ? Est-ce qu'un consensus est trouvé quel que soit `calcul` ?

Exercice 9 ★★ Prouvons maintenant que l'algorithme de la section 4.4.3 résout le consensus distribué en tolérant `f` pannes. On écrit $V1[i]$, $V2[i]$, ...pour la valeur de $V[i]$ dans la machine $1, 2, \ldots$

a. Supposons qu'il y ait un élément de la forme $(3, \cdot)$ dans $V1[f+1]$ mais pas dans $V2[f+1]$. Montrer que la machine 3 a subi une panne. Peut-on préciser à quel point du programme cette panne a du se déclencher ?

b. Soit $2 \leq t \leq f+1$. Prouver que si aucune machine ne tombe en panne entre les lignes 4-7 pour $j = t$, alors à la fin du programme, toutes les machines fonctionnelles ont même ensemble $V[t+1]$.

c. En déduire que pour $t \geq 1$, si deux machines ont des valeurs différentes de $V[t+1]$, alors au moins t machines ont subi une panne.

d. En conclure que l'algorithme ci-dessus résout le consensus en tolèrant `f` pannes.

Notes personnelles

Chapitre 5

Étude de cas : la chaine de blocs de Bitcoin

Cette séance examine en détail la *blockchain* du système Bitcoin. Ce système réalise une monnaie numérique selon les principes généraux décrits à la séance 3. La différence, importante, est que le registre est décentralisé. Cela requiert de résoudre les problèmes de l'élection et du consensus (introduits à la séance 4). La solution mise en œuvre par Bitcoin utilise la *preuve de travail* basée sur du hachage cryptographique.

Une fois ces principes techniques introduits, nous analyserons certains aspects macroscopiques de ce système (niveau de confiance obtenu, niveau de service, impact environnemental) avec une attention particulière à la manière dont ces propriétés sont influencées par des choix techniques en apparence anodins (par exemple le choix d'une fonction de hachage). Un débat éclairé des enjeux de ces systèmes suppose donc une bonne compréhension de ces questions techniques et des domaines scientifiques sous-jacents.

Les objectifs sont que vous sachiez...

- analyser le fonctionnement d'une élection par preuve de travail, et
- expliquer les facteurs qui rendent l'élection par preuve de travail utilisée par Bitcoin catastrophiquement inefficace d'un point de vue énergétique.

5.1 Principes généraux de Bitcoin (et d'autres cryptomonnaies)

Commençons par donner une vue d'ensemble du système Bitcoin. Il s'agit d'une monnaie numérique, similaire en première approximation à celle détaillée en Section 3.1 : la propriété des comptes est attestée par signature électronique, les instruments monétaires sont des mots binaires dont la création et la dépense sont enregistrés dans un registre. Précisons cela...

5.1.1 Comptes et transactions

Chaque instrument monétaire est la propriété d'un **compte**. Concrètement, un compte est une clef de signature cryptographique `(cp,cs)` du type de DSA vu en séance 3. La partie publique, `cp`, sert d'identifiant publique au compte. La partie privée sert à signer les ordres de dépense des instruments financiers possédés par le compte.

Le système Bitcoin utilise une signature ECDSA de courbe elliptique `secp256k1`. Un compte est généralement identifié non pas par sa clef publique, mais par le haché Sha-256 de cette clef publique.

Une **transactions** est un message énonçant le transfert de la propriété d'un instrument monétaire d'un compte A vers un autre compte, voire plusieurs si l'instrument monétaire est fractionable. Pour être **valide**, une transaction doit être *signée cryptographiquement* par A, le propriétaire de l'instrument avant la transaction.

Dans le système Bitcoin, une transaction en bitcoin *détruit* et *crée* un ou plusieurs instruments monétaires, en veillant à ce que les sommes des valeurs nominales des instruments détruits et crées soient égales (pas de création de *monnaie*). Ainsi, la validité d'un instrument monétaire peut être prouvée en exhibant la transaction où il a été créé, et en attestant qu'il n'existe aucune transaction où il a été détruit.

5.1.2 Registre

Un registre recense toutes les transactions valides ayant eu lieu sur le système depuis sa création. Ce registre est :

- *Public* : il est librement disponible sur internet, et n'importe qui peut le télécharger et consulter l'intégralité des transactions qui ont eu lieu dans la monnaie.
- *Distribué* : le registre est maintenu par un système distribué dont chaque machine en stocke une copie. Pour un bon fonctionnement, ce système distribué doit garantir une forme de consensus : lorsqu'une nouvelle transaction est ajoutée au registre, elle doit l'être sur toutes copies (honnêtes).
- *Ouvert* : chaque machine du réseau exécute un algorithme public. Toute machine connectée à internet et exécutant cet algorithme est considérée comme appartenant au réseau. En pratique, plusieurs implémentations de l'algorithme peuvent exister.

Le registre du système Bitcoin est consultable en installant un client lourd bitcoin, ou par exemple via l'interface web `https://blockstream.info/`

Toute transaction est créée par le propriétaire d'une machine et transmise à ses voisines sur le réseau ; la propagation peut être incomplète. Chaque machine dispose d'une réserve (*pool*), qu'elle alimente par les transactions à valider dont elle prend connaissance. Les transactions en réserve *ne sont pas encore* dans le registre.

L'ajout de transactions au registre se fait par phases successives. Chaque phase commence par l'*élection* d'une machine. L'élue constitue alors un *bloc*, formé de transactions tirées de sa réserve. Une fois le bloc constitué, l'élue le transmet au système, qui forme un consensus sur l'acceptation ou le rejet de ce bloc. Ce consensus doit n'accepter que les blocs satisfaisant certaines conditions de validité : chaque transaction doit être correctement signée, doit concerner des instruments monétaires existants et dont le compte signataire est propriétaire, ne doit pas comporter de double dépense, etc. Si le bloc est accepté par le système, chaque machine enlève de sa réserve les transactions du bloc si il y en a, ainsi que des transactions qui seraient incompatibles avec celles qui viennent d'être ajoutées au registre (par exemple parce qu'elles dépenseraient un instrument dont le compte propriétaire vient de changer). Que le bloc soit accepté ou refusé, on commence une nouvelle phase.

Les manières dont le système Bitcoin réalise l'élection et obtient le consensus sont deux points essentiels. Ils sont discutés en détail aux Sections 5.2 et 5.3.

5.1.3 Création monétaire

Chaque bloc ajouté au registre contient, en plus des transactions qu'il ajoute à l'historique, une opération de *création* d'une certaine quantité de nouvelle monnaie, crédités à un compte librement choisi par la machine (élue) qui propose le bloc.

Dans le système Bitcoin, la quantité de monnaie créée à chaque nouveau bloc est déterminée a priori. C'était initialement 50 bitcoins. Ce montant est révisé à la baisse et sera ultimement nul. Autrement dit, la quantité de bitcoin existant est actuellement en augmentation mais finira par être fixée.

5.1.4 Exercices

Exercice 1 ★ Supposons que l'on crée une cryptomonnaie MINECOIN utilisant le même système de signature cryptographique que BITCOIN (ECDSA basé sur la courbe `secp256k1`, cf Section 3.4.2).

a. Qui dispose de l'autorité pour créer un compte valide pour l'une ou l'autre de ces cryptomonnaies ?

b. Suite à un incendie, j'ai irrémédiablement perdu le fichier contenant ma clé secrète et je n'en avais pas fait de copie. Y-a-t'il un moyen de tranférer ou faire transférer les BITCOIN dans un autre de mes comptes dont j'ai gardé la clef secrète ?

c. De peur d'un nouvel incendie, j'ai demandé à des amis de garder chez eux des copies du fichier de ma clé secrète. Je m'aperçoit que quelqu'un d'autre que moi a dépensé mes BITCOINs. M'est-il possible de me faire reconnaître comme le propriétaire originel de ce compte et d'annuler cette transaction ?

d. Une identité créée pour MINECOIN peut-elle être utilisée pour BITCOIN ?

e. Est-ce possible qu'un même compte soit créé plusieurs fois ? Si oui, quelles en seraient les conséquences ? Si non, qu'est-ce qui garantit cette impossibilité ?

Exercice 2 ★ Puisque l'algorithme à exécuter pour faire partie d'un système de type BITCOIN est public, il est tout à fait possible d'en exécuter une version modifiée. Supposons qu'une de mes machines soit élue pour proposer un bloc et que j'en ai modifié le programme pour tricher. Puis-je...

a. ... faire accepter une transaction à partir d'un compte source dont je ne connais pas la clef secrète ?

b. ... décider laquelle de deux transaction en réserve utilisant le même bitcoin (cas de double dépense) sera acceptée ?

c. ... retarder ou annuler une transaction en réserve qui ne me plaît pas ?

5.2 L'idée pour l'élection : la preuve de travail

L'élection nécessaire à la tenue du registre de la chaine de bloc du BITCOIN est réalisée par « preuve de travail ». Voyons cela...

5.2.1 Exemple introductif : lutter contre le le déni de service

Une attaque par *déni de service* consiste à saturer une machine de demandes (inutiles) afin de la rendre indisponible aux utilisateurs légitimes. Ce type d'attaque vise notamment les éléments d'un réseau assurant certaines tâches clef (authentification des utilisateurs, transmission de messages, ...). L'idée de la preuve de travail a été introduite par Dwork et Naor au début des années 1990 pour lutter contre les attaques par déni de service. Cette idée consiste à ce que la machine fournissant un service exige, avant de traiter une demande, que le requiérant effectue un travail pour elle. Ce travail n'a d'autre finalité que... de faire travailler le requiérant, aussi on parle de *résoudre un puzzle*. Si la difficulté du puzzle est bien dosée, l'effort qu'il demande sera négligeable pour un utilisateur normal et écrasant pour un utilisateur effectuant un nombre abusif de requêtes.

Cette idée demande des puzzles faciles à construire et à vérifier, et dont on peut contrôler le temps effectivement nécessaire à la résolution (ni trop, ni trop peu). Les fonctions de hachage cryptographiques sont là encore utiles. Choisissons-en une, par exemple SHA-256, et fixons deux paramètres $\ell > k$. On tire au hasard un mot binaire w de longueur ℓ, on transmet à l'utilisateur les k premiers chiffres de w, l'entier ℓ et le haché $y = \text{SHA-256}(w)$; le puzzle consiste à trouver un mot binaire w' de longueur ℓ de haché y. L'utilisateur peut résoudre cela en testant les $2^{\ell-k}$ complétions possibles du préfixe donné, mais peut difficilement faire mieux en raison du caractère cryptographique de la fonction de hachage. [1]

1. Ce point reste bien entendu à formaliser par une réduction du type de la Proposition 1 ou du Théorème 4.

5.2.2 Contexte de l'élection dans Bitcoin

Soulignons quelques différences entre l'élection que doit organiser le système Bitcoin et le problème étudié au Chapitre 4.

Le réseau sur lequel Bitcoin est exécuté est *pair à pair*. Il autorise toute machine à le rejoindre et à le quitter, à tout moment. L'algorithme doit donc fonctionner sur un réseau de topologie non seulement arbitraire, mais aussi changeante au fil du temps.

Il y a un double enjeu à gagner l'élection. Pour un participant honnête, rédiger un bloc permet de choisir le compte propriétaire des bitcoins nouvellement créés. Pour un participant malhonnête, cela ouvre de plus la possibilités de manipulations (par exemple de fraude à la double dépense évoquée à l'exercice 2 ci-dessus). Soulignons que le système Bitcoin ne permet pas de contrôler les algorithmes effectivement exécutés par les machines du réseau. Il est donc important que le protocole d'élection empêche toute triche *y compris par modification de l'algorithme*. Cela disqualifie les approches par identifiant unique présentées au Chapitre 4.

5.2.3 Élection distribuée ouverte par preuve de travail

Une élection par preuve de travail consiste à proposer un même puzzle à l'ensemble des machines, et à élire la première machine qui réussit à le résoudre. On souhaite bien entendu que le puzzle ne puisse être résolu que par force brute. Autrement dit :

> Un processeur = 1 voix.

Rien n'empêche un participant de biaiser l'élection en sa faveur, en faisant travailler plusieurs machines. Soulignons, cependant, que l'effort requis pour biaiser significativement l'élection est proportionnel à la taille du système :

> Pour avoir $x\%$ de chances de gagner l'élection, il faut contrôler $x\%$ de la puissance de calcul du système.

Soulignons que contrôler une puissance de calcul se traduit par un **coût physique**, puisqu'il faut fabriquer les machines, puis les alimenter voire les refroidir pendant qu'elles s'efforcent de résoudre le puzzle.

5.2.4 L'élection dans Bitcoin : bloc, nonce et minage

Dans le système Bitcoin, la rédaction du bloc précède l'élection de son rédacteur. C'est en effet la rédaction du bloc à ajouter au registre qui constitue le puzzle de la preuve de travail : pour être considéré comme valide, un bloc doit satisfaire une condition de hachage par Sha-256 que l'on ne sait résoudre que par force brute. Examinons cela en détail.

Structure d'un bloc du registre Bitcoin

Le registre Bitcoin est une chaîne de blocs. Chaque bloc comporte un en-tête et une liste de transactions. Le format de description d'une transaction nous importe peu[2]. L'en-tête est un mot binaire de 80 octets qui se décompose comme suit :

- 4 octets donnant la version du protocole Bitcoin auquel ce bloc annonce se conformer,
- 32 octets donnant le haché de l'en-tête du bloc précédent dans le registre,
- 32 octets donnant le haché de la racine de l'arbre de Merkle des transaction du bloc,
- 4 octets donnant l'heure approximative de création de ce bloc,
- 4 octets donnant la difficulté cible,
- 4 octets libres formant un mot 32 bits appelé nonce.

Ainsi, l'en-tête d'un bloc détermine (par hachage) l'en-tête du bloc précédent et (par hachage d'arbre de Merkle) la liste des transactions que ce bloc contient.

2. On peut en voir des exemples à `https://blockstream.info`.

Hachage et difficulté

Pour qu'un bloc B soit considéré valide, il doit satisfaire notamment [3] les conditions suivantes :

(i) le « haché de l'en-tête du bloc précédent » doit correspondre à un bloc B' valide du registre,

(ii) les transactions contenues dans B doivent être correctement signées et consommer des bitcoins qui ont été créés mais pas dépensés par la chaîne de blocs se terminant à B',

(iii) le haché par SHA-256 du mot 640-bits formant l'en-tête de B doit être inférieur à la difficulté indiquée.

Voici, pour être concret, quelques unes des valeurs (en hexadécimal) prises par la difficulté au fil du temps, mesurée en nombre de blocs :

n° de bloc	cible
30 000	00000000ffff0000000000000000000000000000000000000000000000000000
50 000	000000002a1115000000000000000000000000000000000000000000000000
100 000	00000000004864c000000000000000000000000000000000000000000000000
200 000	00000000000005db8b000000000000000000000000000000000000000000000
300 000	0000000000000000896c000000000000000000000000000000000000000000
400 000	000000000000000006b99f0000000000000000000000000000000000000000
500 000	00000000000000000964500000000000000000000000000000000000000000
600 000	0000000000000000015a35c000000000000000000000000000000000000000
671 000	0000000000000000000cf4e300000000000000000000000000000000000000

Remarquons que pour être inférieur à la valeur cible autour du bloc n° 600 000, le haché doit nécessairement commencer par $18 \times 4 + 3 = 75$ zéros. Si l'on considère que les valeurs hachées par SHA-256 sont distribuées uniformément dans les mots 256 bits, la probabilité qu'un bloc assemblé sans plus de soin satisfasse ce critère est inférieure à 2^{-75}. On revient en Section 5.3.3 sur les critères d'évolution de cette difficulté.

Minage

La définition d'un bloc dans la blockchaine de BITCOIN ne laisse de liberté que sur le choix des transactions à inclure (y compris leur position dans l'arbre de Merkle) et sur le nonce. Assigner telle ou telle valeur au nonce a pour seul intérêt de modifier le haché du bloc. Il est communément admis que les propriétés cryptographiques de SHA-256 assurent que l'on ne peut produire un bloc valide que par force brute. En schématisant (fortement), la recherche d'un bloc valide ressemble donc à :

```
1 def mine():
2     choisir des transactions valides
3     remplir l'en-tête E[0..75] en conséquence
4     nonce = 0 et E[76..79] = 0
5     while (hash(E) > cible):
6        nonce++ et E[76..79] = nonce
```

Les machines du système BITCOIN qui cherchent à résoudre un tel puzzle sont appelées des *mineurs*. Dès qu'un mineur a résolu un tel puzzle, il transmet le bloc obtenu (on dit *miné*) au réseau.

3. Il y a diverses autres conditions plus élémentaires : la version du protocole doit être valide, l'arbre de Merkle des transactions doit bien avoir pour haché la valeur indiquée, la difficulté indiquée doit être celle du réseau au moment où le bloc est trouvé, etc.

5.2.5 Exercices

Exercice 3 ★★ Proposez un système pour lutter contre la diffusion de pourriels (*spams*) à base de preuve de travail.

Exercice 4 ★ Notons c_0 la valeur cible requise au bloc 30 000. Existe-t-il, pour tous E[0..75] une valeur du nonce E[76..79] qui assure que hash(E) $\leq c_0$?

Exercice 5 ★ Déterminez à l'aide de `https://blockstream.info/` le numéro d'un bloc miné récemment, son nombre de transactions, son nonce et le nombre de bitcoin crées dans ce bloc.

5.3 L'idée pour le Consensus : une chaîne dans un arbre

Lorsqu'un mineur propose un bloc, les machines du système Bitcoin doivent décider si elles l'acceptent ou pas. Ce problème peut sembler simple (« accepter le bloc ssi il est valide »), sauf que rien n'empêche deux mineurs de réussir à former des blocs valides succédant *au même bloc* à peu près au même moment. Dans une telle éventualité, au plus un de ces blocs peut être accepté. Les machines du système doivent en principe former un consensus sur ce choix. Voyons comment cela fonctionne.

5.3.1 Diffusion d'un nouveau bloc dans le réseau

Lorsqu'une machine réussit une opération de minage, elle diffuse le nouveau bloc produit aux autres machines du système Bitcoin. Chaque autre machine vérifie que le bloc est correct et décide, localement, de l'accepter ou de le refuser. S'il est accepté, le bloc est ajouté comme prolongement du bloc pré-existant qu'il annonce étendre.

Si on suppose que la diffusion et l'acceptation/rejet d'un bloc par le système est instantanée, cette méthode produit une chaîne de blocs. En effet, lorsqu'un mineur accepte un nouveau bloc, il recommence son minage à partir de ce bloc. (Il met aussi à jour sa réserve en supprimant les transactions contenues dans et incompatible avec le bloc récemment ajouté.) Dans ce cas idéal, il faudrait que deux mineurs réussissent *simultanément* à miner un bloc pour que l'on ait un problème : le réseau *dans son ensemble* devrait choisir l'un ou l'autre nouveau bloc.

La propagation d'un bloc dans le réseau n'étant pas instantanée, la possibilité qu'à un moment donné deux nouveaux blocs propagés étendent le même bloc préexistant est réelle. Il suffit pour cela que l'intervalle de temps séparant deux minages réussis soit inférieure au temps de diffusion entre ces deux mineurs sur le réseau. Le besoin de former un consensus est donc réel.

5.3.2 Une chaine dans un arbre

L'approche du système Bitcoin pour obtenir un consensus peut être résumé par « que le plus long gagne ! ». En détail :

a. Chaque machine accepte tout bloc correct. Ainsi, si deux blocs étendent correctement un bloc pré-existant, les deux sont mémorisés. Chaque machine maintient donc non pas une *chaîne*, mais un *arbre*.

b. La *chaîne principale* d'un arbre est le chemin *le plus long* dans cet arbre, partant de la racine. Plus précisément, la longueur du chemin est calculée en pondérant chaque bloc par sa difficulté de minage. Les autres chemins maximaux pour l'inclusion et partant de la racine sont appelés *chaînes secondaires*.

c. Chaque machine définit le *registre* comme la suite des transactions enregistrées dans la chaîne principale de son arbre.

d. Le système Bitcoin est en *état de consensus* si toutes les machines ont la même chaîne principale, et donc même état du registre.

Lorsque le système Bitcoin n'est pas en état de consensus, différentes machines ont des idées différentes sur qui possède quoi. Lorsque cette divergence se résout, certaines machines changent de chaîne principale et donc de registre. Du point de vue de ces machines, certaines transactions sortent du registre pour retourner en réserve ou disparaître : elles sont « invalidées ».

5.3.3 Auto-ajustement de la difficulté

La fréquence à laquelle un nouveau bloc est miné est un paramètre critique pour le fonctionnement du système Bitcoin. Si cette fréquence est trop faible, les transactions peinent à être validées et le système s'engorge. Si cette fréquence est trop forte, les chaînes secondaires se multiplient et le système passera une grande partie du temps hors état de consensus.

Le système Bitcoin vise que la fréquence de minage soit autour d'un nouveau bloc toutes les 10 minutes ; ce choix correspond à un régime où l'état de consensus est fréquent. En pratique, la fréquence moyenne de minage est déterminée par deux paramètres : la puissance de calcul de l'ensemble des mineurs et la difficulté du minage, c'est à dire la valeur cible. Tous les 2016 blocs, le système compare le temps réellement pris pour miner ces blocs à la moyenne théorique visée (ici, 14 jours) et *ajuste la cible en conséquence* : si les blocs sont minés trop vite, on abaisse encore la cible (ce qui augmente la difficulté), sinon on la réhausse.

5.3.4 Exercices

Exercice 6 ★★ On vous présente les principes d'un projet de blockchaine ouverte :

- les participants s'identifient par signature cryptographique,
- pour participer au tirage concernant la rédaction du bloc N, un participant doit avoir été déclaré avant ce tirage,
- on constitue une liste des participants déclarés par ordre lexicographique de clef publique, puis on utilise le haché de cette liste comme "source de bits aléatoires" pour produire une permutation pseudo-aléatoire des participants,
- chaque participant est libre de rédiger un bloc et de le proposer,
- chaque nœud ajoute chaque bloc qui lui semble correct, avec un poids 2^{-r} où r est le rang de l'auteur du bloc dans la permutation produite,
- la blockchaine est le chemin dont la somme des poids est maximale.

Identifiez au moins une vulnérabilité de ce système et expliquez pourquoi Bitcoin n'est pas vulnérable à cette attaque.

Exercice 7 ★★ (Avec machine) Commençons par examiner le nombre de transactions enregistrées par unité de temps dans la blockchain du Bitcoin. On utilise pour cela https://www.blockchain.com/charts/n-transactions.

a. Commençons par vérifier le format des données.
   (a) Positionnez la courbe noire sur "*Confirmed transactions per day*" et désactivez la courbe bleue.
   (b) Comment ces données sont-elles obtenues ? Cela vous semble-t-il fiable ?
   (c) Téléchargez les données sur 1 an et déterminez la fréquence des mesures.
   (d) Téléchargez les données complètes ("*all*") et vérifiez que vous y retrouvez les données sur 1 an.
b. Déterminez le nombre moyen de transaction pour l'année 2018.
c. Déduisez-en le nombre moyen de transactions confirmées chaque mois par Bitcoin en 2018 et le nombre de transactions confirmées au total en 2018.

d. Supposons que l'intégralité des transactions en Bitcoin en 2018 soient consacrées aux habitants de la Communauté Urbaine du Grand Nancy. Combien de transactions par jour est-ce que cela représente par habitant ?

e. Même question pour la région Grand Est.

Exercice 8 ★★★ (Avec machine)

a. À partir de `https://www.blockchain.com/explorer/charts/difficulty` déterminez la difficulté minimale du minage au cours de l'année 2018.

b. Supposons que le calcul d'un haché par Sha-256 revient à tirer aléatoirement un mot 256 bits selon la loi uniforme. Quel est, en fonction de d, le nombre moyen de hachés qu'il faut calculer pour en trouver un qui satisfait à la difficulté d ?

c. Quel minorant ce modèle probabiliste fournit-il pour le nombre de hachés calculés pour chaque bloc ?

d. Quel minorant ce modèle probabiliste fournit-il pour le nombre moyen de téra-hash calculés par seconde (TH/s) en 2018 ?

5.4 Consommation énergétique

L'impact environnemental du système Bitcoin est une question importante. Pour l'examiner avec un peu de soin [4], nous allons nous appuyer sur une *analyse de cycle de vie* (ACV) de Köhler et Pizzol [2]. L'ACV est une méthode établie d'analyse multicritère de l'impact environnemental d'un système ; on en donne une introduction très sommaire en Annexe C.

5.4.1 Préambule : industrialisation de la preuve de travail

À ce jour, le caractère cryptographique de Sha-256 n'a pas été compromis. La recherche d'un bloc de haché inférieur à la cible se fait donc en essayant des blocs les uns après les autres. La puissance de calcul dédiée au système Bitcoin peut donc se mesurer en "nombre de hachés calculés par seconde". La figure 5.1 révèle que l'évolution de cette puissance au fil du temps n'est pas régulière (noter l'échelle

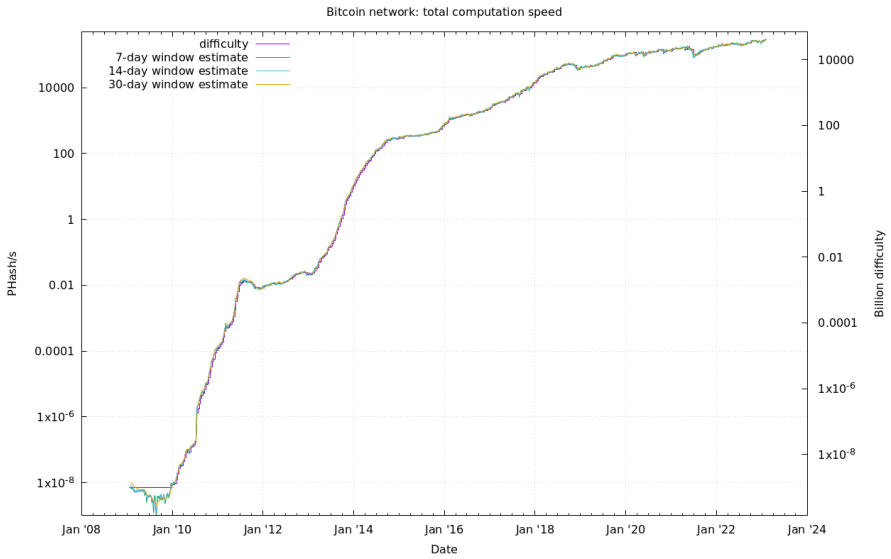


Figure 5.1 – Évolution du nombre de hachés calculés par seconde (1 PHash/s = 10^{15} hash/seconde). L'échelle est semi-logarithmique. (Source : `http://bitcoin.sipa.be/`.)

semi-logarithmique). Notons en particulier qu'elle a été multipliée par 10^6 entre début 2010 et mi-2011,

4. Une simple recherche sur internet permet de trouver facilement des sources à ce sujet, mais les données qu'elles fournissent peuvent s'avèrer discutables (analyse tronquée, approximations ou extrapolations grossières, etc).

puis à nouveau par 10^4 entre début 2013 et mi-2014. Ces deux phases coïncident avec des changements technologiques : le minage sur GPU se développe à partir de 2010, et le minage sur ASIC se développe à partir de 2013.

Pour comprendre ces évolutions, il est utile de faire un (très bref) point d'architecture. L'écrasante majorité des processeurs actuels (CPU, GPU, TPU, ...) sont des assemblages de transistors, qui composent des portes logiques (AND, OR, NOT, ...) qui forment elle-même des circuits réalisant des opérations élémentaires (addition, copie, comparaison, ...) et des variables élémentaires. Ces opérations élémentaires constituent le langage natif du processeur, ce qu'on appelle son *assembleur* ; les variables internes au processeur sont ce que l'on appelle ses *registres*. Tout programme exécuté sur un tel processeur doit être traduit en assembleur. [5] et toute donnée manipulée transite par un registre. Le coût d'exécution d'une instruction assembleur provient d'une part de son traitement (par exemple réaliser effectivement l'addition de deux registre) et d'autre part de sa préparation (par exemple charger les données à traiter dans des registres) et de sa gestion (par exemple, identifier l'instruction suivante à exécuter).

Un GPU se distingue d'un CPU en ce que ses registres ne stockent pas une seule valeur, mais un *vecteur* de valeurs de taille typiquement assez grande [6]. Les coûts de gestion sont ainsi réduits, puisqu'une seule instruction opère sur beaucoup de données. Cela ne vient pas sans contrainte, puisqu'il faut que le traitement de données que l'on souhaite réaliser puisse s'écrire sous la forme d'une suite d'opérations vectorielles. Il s'avère que Sha-256 peut être efficacement évaluée en parallèle par un tel calcul vectoriel, d'où le succès de son déploiement sur GPU. On peut imaginer que d'autres fonctions soient moins adaptées au calcul sur GPU.

Un processeur, CPU ou GPU, est un système très complexe et pensé pour traiter très efficacement un large éventail de tâches. Il se trouve que le calcul, en boucle, de fonctions Sha-256 sous-utilise très largement ces capacités, et peut se faire sur un circuit intégré dédié bien plus simple. C'est ce que l'on appelle un ASIC. Comme pour le passage sur GPU, le passage sur ASIC ne va pas de soi, et doit se faire pour chaque fonction de hachage spécifiquement. Il s'avère aussi que Sha-256 est facilement calculable sur ASIC. On peut imaginer que d'autres fonctions le soient moins. Par ailleurs, un ASIC doit être conçu et fabriqué sur mesure pour la tâche à réaliser, ce qui n'est rentable que si cela permet des économies d'échelle substantielles.

5.4.2 Type d'ACV et unité fonctionnelle

Köhler et Pizzol proposent une ACV-A (attributive) rétrospective portant sur l'année 2018 et une ACV-C (conséquentielle) prospective portant sur la croissance du système au-delà de 2019 (année de l'étude). L'ACV-C comporte trois scénarii.

L'*unité fonctionnelle* de l'ACV est "le calcul d'1 TH de Sha-256". Un TH est un Terahash, c'est à dire 10^{12} valeurs hachées ; l'étude estime, sur la base de l'évolution de la difficulté de minage, qu'il a été calculé de l'ordre d'un milliard de TH en 2018. Plus précisément, l'ACV-A détermine le coût moyen de calcul d'1 TH en 2018 et les ACV-C déterminent le coût moyen, dans divers scénarii, de calcul d'1 TH post-2019 (année de publication de l'étude).

5.4.3 Inventaire

La *phase d'inventaire* porte sur les machines et sur l'énergie alimentant ces machines.

L'inventaire des machines combine un recensement des grands types de machines et l'ACV d'un ordinateur-type, au prorata du poids. Pour l'ACV-A, le recensement est fait sur la base des blocs ayant été minés en 2018, et distingue trois types de machines qui représentent ensemble 95% des minages. Pour l'ACV-C, les trois scénarii prolongent la répartition de 2018 ("Business as usual" et "Localisation") ou l'améliore ("Technologie").

L'inventaire de l'énergie s'appuie sur un recensement de la répartition géographique des mineurs et une modélisation des mix énergétiques locaux. Ce recensement est fait au niveau de granularité

5. C'est précisément cette traduction qu'assure l'interpréteur `python`.
6. Des vecteurs de 16 384 ou 32 768 entrées ne sont pas rares.

d'une région : les 53.5% des mineurs localisés en Chine ne sont pas supposés utiliser le mix énergétique national, mais sont traités par région (soit 30.5% dans le Sichuan, 12.3% en Mongolie intérieur, et 10,7% dans le Xinjiang). Pour l'ACV-C, les trois scénarii prolongent la répartition de 2018 ("Business as usual" et "Technologie") ou l'améliore en favorisant les sites où le mix est moins carbonné ("Localisation"").

5.4.4 Traduction

Köhler et Pizzol traduisent l'impact environnemental en une douzaine d'indicateurs (*c.f.* Table 2 de l'article et début de la section *results and discussion*). Pour la discussion, nous allons ici retenir deux résultats de l'ACV-A :

a. Le coût énergétique est estimé à 27 mWh par TH.
b. Le potentiel de réchauffement climatique selon la méthode IPCC (GWP IPCC) est de 15 mg CO_2-équivalents par TH.

5.4.5 Exercices

Exercice 9 ★★ Revisitons les résultats des exercices 7 et 8 à la lumière des résultats de l'ACV de Köhler et Pizzol.

a. Calculez les nombres suivants :
   - `ns` le nombre de secondes par an,
   - `nths` le nombre moyen de térahash calculés par seconde en 2018
   - `nth` le nombre total de térahash calculés en 2018
b. Déduisez-en une estimation de l'énergie totale `etot` consommée par le réseau Bitcoin en 2018.
c. Déduisez-en une estimation du potentiel de réchauffement climatique selon la méthode IPCC (GWP IPCC) `potrec` émis par le réseau Bitcoin en 2018.

Exercice 10 ★

a. Quel a été, en 2018, le coût énergétique moyen (en kWh) de la validation d'une transaction ?
b. Quel a été, en 2018, le potentiel de réchauffement climatique moyen (en kg CO_2-équivalents) de la validation d'une transaction ?

Exercice 11 ★★

a. Comment est-ce que le nombre de transactions validées par seconde a évolué entre 2018, 2019 et 2020 ?
b. Comment est-ce que le nombre de hachés par seconde a évolué entre 2018, 2019 et 2020 ?
c. Comment est-ce que le coût énergétique moyen et le potentiel de réchauffement climatique moyen de la validation d'une transaction a évolué entre 2018, 2019 et 2020 ?

La discussion ci-dessus devrait faire apparaître le coût environnemental d'une décentralisation « à la Bitcoin » comme **prohibitivement élevé**. Un tel coût n'est pas une caractéristique intrinsèque aux chaînes de blocs, mais devrait disqualifier l'usage des preuves de travail pour résoudre le problème de l'élection.

5.5 Pistes d'approfondissement

On conclue cette séance par quelques pistes d'approfondissement.

5.5.1 Dynamique d'une cryptomonnaie

Il est utile d'expliciter une dynamique importante sous-jacente au bon fonctionnement du système Bitcoin, valide pour d'autres cryptomonnaies.

L'adoption du Bitcoin comme monnaie par des acteurs économiques repose sur la confiance qu'ils accordent à sa *fiabilité* (par exemple la non-réversibilité des transactions). La fiabilité du Bitcoin repose en partie sur le fait que les mineurs sont *diversifiés* (un individu réussissant à miner une fraction importante des blocs pourrait annuler une transaction). Une large diversification des mineurs repose sur le fait que l'opération de minage est, pour eux, *rentable*. Le coût du minage est certain (achat d'une machine, alimentation en énergie, connection au réseau, refroidissement, entretient, ...) et en monnaie reconnue, sa récompense est incertaine (seule la réussite est rémunérée) et en bitcoin. Ainsi, la rentabilité du minage pour une personne repose sur l'adoption du Bitcoin comme monnaie par des acteurs économiques et sur une fréquence de réussite raisonnable.

Soulignons la circularité de ces implications : l'adoption du Bitcoin comme monnaie dépends de conditions qui dépendent, à leur tour, de l'adoption du Bitcoin comme monnaie. Il y a là une boucle de rétroaction. Le système Bitcoin a connu au moins deux types d'équilibres différents. En 2009, il n'était pas adopté comme monnaie et personne ne le minait. En 2020, il est en partie adopté comme monnaie et le minage peut être rentabilisé et professionalisé. Le passage d'un type d'équilibre à l'autre n'a rien d'évident, comme l'illustre par exemple l'Auroracoin.

5.5.2 Sécurité et confiance

Quelle confiance peut-on avoir dans la résistance du registre du système Bitcoin à la fraude ? Précisons trois points. D'une, on suppose que les propriétés cryptographiques de Sha-256 et de la version d'ECDSA utilisée par le système Bitcoin ne sont pas compromises. Si cela devait être le cas, le système perdrait largement en fiabilité. De deux, on ne parle pas de la confiance en l'utilisation de Bitcoin comme monnaie. Cette question est intéressante mais déborde largement du cadre de ce cours (ne serait-ce qu'en raison du caractère largement spéculatif de l'utilisation de cette monnaie). De trois, on ne parle pas des risques d'usurpation de compte par vol de clef privée, problématique de cybersécurité.

Ces trois points précisés, un des principaux risques restant est lié à la nature probabiliste du consensus dans le système Bitcoin : un groupe d'individus agissant de concert et controlant une fraction significative de la puissance de calcul du système serait en mesure d'annuler une transaction (récente). En effet, statistiquement, une fraction significative des blocs ajoutés à la blockchaine devrait être produite par des machines contrôlées par ce groupe. Ce groupe pourrait par exemple choisir à quelle branche s'ajoutent leurs blocs, et par là exclure le dernier bloc miné (en cherchant à miner un bloc faisant suite à l'avant-dernier bloc miné).

La non-réversibilité des transactions repose donc en partie sur le fait que la communauté des mineurs soit *faiblement concentrée*. Dans un système économique, le taux de concentration est influencé par l'*intensité capitalistique* de l'activité. En effet, s'il faut immobiliser un capital conséquent pour être compétitif, les dynamiques classiques de marché peuvent conduire à une concentration des ressources de minage entre quelques mains.

> Le passage du minage du CPU au GPU puis aux ASIC a induit une augmentation de l'intensité capitalistique, et a donc contribué à augmenter le risque de concentration d'une forte part du minage entre quelques mains.

Ainsi, un paramètre purement technique comme le choix de la fonction de hachage cryptographique utilisée dans le système de preuve de travail peut avoir une influence déterminante sur l'intensité capitalistique du minage et, in fine, la confiance que la blockchaine inspire. À titre de comparaison, la cryptomonnaie Ethereum utilise une preuve de travail basée sur une fonction de hachage pour laquelle la conception d'ASIC se révèle peu rentable (on la dit "ASIC-résistante").

5.5.3 Décentralisation ?

Il est souvent mis en avant que le système Bitcoin a la vertu d'être décentralisé. Si le système Bitcoin est conçu comme un système décentralisé et ouvert on peut remarquer les points suivants :

a. Comme signalé, l'évaluation de la fonction Sha-256 a donné lieu à une course technologique (CPU, puis GPU, puis ASIC), augmentant l'intensité capitalistique du secteur économique de minage. Il s'en suit une concentration du minage en un faible nombre d'acteurs.

b. Un système tel que Bitcoin est occasionnellement amené à faire évoluer certaines de ses caractéristiques. On peut consulter les *bitcoin improvement proposals* à

   https://github.com/bitcoin/bips

   Ces évolutions s'incarnent au travers de modifications du code des clients utilisés sur le réseau. (Cela peut, dans certains cas, conduire à une séparation du réseau en deux ; voir par exemple Bitcoin Cash.) Proposer des évolutions demande une compétence technique amenant là encore à une restriction du nombre d'acteurs.

Ainsi, une conception décentralisée n'implique ni *fonctionnement* ni *gouvernance* décentralisés.

5.6 Références bibliographiques

[2] Susanne Köhler and Massimo Pizzol. Life cycle assessment of bitcoin mining. *Environmental science & technology*, 53(23) :13598–13606, 2019.

Notes personnelles

Chapitre 6

Algorithmes de consensus

Les instances de consensus distribué asynchrone se posent très fréquemment en pratique, par exemple dans les grandes bases de données. Le chapitre précédent a établi que ce problème est impossible à résoudre parfaitement. Ce dernier chapitre aborde les solutions pratiques, en présentant deux algorithmes de consensus. Ces algorithmes sont nécessairement imparfaits, puisque le problème est impossible à résoudre parfaitement. Ils sont aussi présentés ici de manière assez épurées, et leur déploiement demande encore un travail substantiel d'ingéniérie algorithmique et logicielle.

Les objectifs sont que vous compreniez les principes de conception et d'analyse d'algorithmes de consensus formant la base des solutions déployées et en cours de déploiement.

6.1 Algorithme Paxos pour le consensus asynchrone avec pannes

De nombreuses solutions industrielles de consensus [1] sont basées sur l'algorithme Paxos développé par Lamport à la fin des années 1990. Rappelons le problème dont il est question ici :

Consensus tolérant aux pannes

Entrée : Chaque machine a un vote initial valant 0 ou 1.

Sortie : Chaque machine choisit un vote final valant 0 ou 1. Ces votes doivent satisfaire les conditions de cohérence globale suivantes :

a. Toutes les machines fonctionnelles ont le même vote final.

b. Le vote final égale l'un des votes initiaux.

6.1.1 Le cadre

L'algorithme Paxos opère sur un système asynchrone à réseau complet et identifiants uniques. Nous allons établir que lorsque Paxos termine, la solution qu'il donne est correcte, mais que pour toute entrée, il existe un ordonnancement des pas de calculs qui empêche l'algorithme de terminer. Autrement dit, l'algorithme peut ne pas terminer mais quand il termine, il est correct.

Paxos est constitué de trois sous-algorithmes indépendants :

- Les **proposeurs** (*proposers*) peuvent proposer une valeur de consensus.
- Les **accepteurs** (*acceptors*) peuvent accepter une valeur de consensus.
- Les **scribes** (*learners*) peuvent prendre acte qu'un consensus a été trouvé.

Ces trois rôles échangent des messages entre eux. Ces messages comportent généralement un ou deux paramètres :

- Un *nombre* n, qui est simplement un entier qui augmente à mesure que l'algorithme progresse. On peut considérer n comme une mesure du temps, à ceci près l'asynchronisme empêche toute mesure commune du temps.

1. *C.f.* https://en.wikipedia.org/wiki/Paxos_(computer_science)#Production_use_of_Paxos

- Une *valeur* (généralement notée v), qui est une proposition de vote final. Dans notre cas, ce sera toujours 0 ou 1. On peut envisager des problèmes de consensus plus généraux, par exemple pour s'accorder sur un tarif du KW/h dans un réseau électrique local intelligent.

Chaque machine du système peut jouer un ou plusieurs rôles, on parle donc de *processus*. (En cas de confusion, on peut supposer que chaque machine joue exactement un des rôles, c'est à dire exécute un seul processus.) L'algorithme est prévu pour un système *fermé*, donc on peut supposer connu le nombre de processus de chaque rôle.

6.1.2 L'algorithme, en résumé

Lamport [4] résume ainsi son algorithme Paxos :

Phase 1.

a. A proposer selects a proposal number n and sends a prepare request with number n to a majority of acceptors.

b. If an acceptor receives a prepare request with number n greater than that of any prepare request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal (if any) that it has accepted.

Phase 2.

a. If the proposer receives a response to its prepare requests (numbered n) from a majority of acceptors, then it sends an accept request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

b. If an acceptor receives an accept request for a proposal numbered n, it accepts the proposal unless it has already responded to a prepare request having a number greater than n. Whenever an acceptor accepts a proposal, it sends a message to all learners, sending them the proposal.

Précisons les points suivants :

a. Si un proposeur ne reçoit pas de réponse de la part d'un accepteur dans un délai fixé (appelé *timeout*), il considère que la réponse est négative.

b. Si un proposeur échoue à obtenir une majorité de réponses positives à une *prepare request*, il recommence en 1a avec un n strictement plus grand.

c. Un proposeur peut abandonner une proposition en cours, à n'importe quel moment. Pour cela, il suffit qu'il initie une nouvelle proposition et ignore toute réponse à la précédente.

d. Le comportement décrit ci-dessous pour un proposeur est en fait celui de *chaque* proposeur. Autrement dit, les proposeurs sont en compétition pour obtenir les engagements des accepteurs.

e. Un accepteur peut accepter plusieurs propositions de choix.

f. Le fait qu'il y ait plusieurs accepteurs n'est utile que pour résister aux pannes : en l'absence de pannes, Paxos fonctionne correctement avec un unique accepteur.

g. Lorsqu'un scribe apprend que plus de la moitié des accepteurs ont accepté une valeur v, il reporte que cette valeur v comme résultat du consensus.

6.1.3 Exercices

Exercice 1 ★★ Listez les points de Paxos qui ne vous semblent pas clairs ou pas complètement spécifiés (il y en a plusieurs).

Exercice 2 ★★★ La description par phases entremêle ce que font les différents rôles. Séparez cela en donnant pour chaque rôle l'algorithme qu'il doit exécuter. *(Attention, ceci représente un travail substantiel et demande de bien comprendre l'algorithme.)*

Exercice 3 ★★ Décrivez une exécution dans laquelle des messages sont échangés mais les scribes n'enregistrent aucun progrès dans la formation d'un consensus.

Exercice 4 ★★★ On établit maintenant la propriété suivante pour tous $n_0 < n_1$:

> $P(n_0 \to n_1)$: si une majorité d'accepteurs accepte une *accept request* de *proposal number* n_0 et de valeur v_0, alors toute *accept request* émise avec un *proposal number* $n_1 > n_0$ est aussi de valeur v_0.

Notez que $P(n_0 \to n_1)$ ne fait aucune hypothèse sur l'ordre chronologique dans lequel les deux *accept request* sont émises.

a. Considérons un accepteur A qui accepte, à un instant T, une *prepare request* de *proposal number* n. Est-ce que le « choix accepté par A de plus grand *proposal number* $< n$ »peut changer *après* T ? Justifiez votre réponse.

b. Prouvez $P(n_0 \to n_0 + 1)$.

c. Prouvez $P(n_0 \to n_1)$ pour tous $n_0 < n_1$.

Exercice 5 ★★

a. Supposons que l'on ait déployé Paxos sur un système formé de $p + a + s$ machines distinctes, chaque machine remplissant un seul rôle : il y a p proposeurs, a accepteurs et s scribes. Combien de pannes de chaque type de machines Paxos supporte-t-il ?

b. Supposons maintenant que l'on ait déployé Paxos sur un système formé de m machines distinctes, chaque machine remplissant *les trois* rôles. Combien de pannes Paxos supporte-t-il ?

c. Proposez un ensemble d'informations, aussi minimal que possible, dont l'archivage permet une panne avec redémarrage sans perturber l'exécution de Paxos.

Commentaire

La propriété $P(n_0 \to n_1)$ ci-dessus implique qu'à partir du moment où un proposeur a réussi à faire accepter à une majorité d'accepteur le choix d'une valeur v lors d'une *accept request*, deux propriétés sont vraies :

(i) Un accepteur ayant choisi v ne change plus d'avis. (Notons, cependant, qu'il peut accepter de nouvelles *accept request* de *proposal number* plus grand, mais ces requêtes doivent avoir même valeur v.)

(ii) Un accepteur n'ayant pas choisi v ne reçoit plus de proposition de choisir autre chose que v.

Ainsi, si l'algorithme progresse au sens ou les accepteurs continuent à recevoir des *accept request*, alors il progresse forcément vers un consensus.

6.2 Algorithme BBA* pour le consensus synchrone avec pannes Byzantines

L'algorithme BBA* a été proposé par Silvio Micali en 2017 et est intégré à la chaîne de blocs Algorand qu'il développe [3]. La présentation que l'on suit est tirée de ses notes [5]. Rappelons le problème de Consensus Byzantin dont il est question ici :

Consensus Byzantin

Entrée : Chaque machine a un vote initial valant 0 ou 1

Sortie : Chaque machine choisit 0 ou 1. Ces choix doivent satisfaire les conditions suivantes :

a. Toutes les machines fiables font le même choix.

b. Si toutes les machines fiables ont le même vote initial x, leur choix final est x.

6.2.1 Le cadre

L'algorithme BBA* opère sur un système synchrone à n machines, à réseau complet muni d'une fonction de hachage cryptographique H et d'un système de signature numérique (Schnorr, DSA, ECDSA, ...). Pour simplifier la présentation, on suppose que $n = 3t + 1$ et on identifie chaque machine avec sa clef publique (supposée unique). Chaque machine connaît les clefs publiques des autres machines (et en particulier leur nombre).

L'algorithme tolère des pannes Byzantines « de complexité polynomiale ». Plus précisément, on suppose qu'un Adversaire peut corrompre des machines, leur faire exécuter un code arbitraire et plus généralement coordonner leurs actions. Une machine est fiable si elle n'a pas été corrompue. L'Adversaire connaît les clefs publiques de toutes les machines et les clefs secrètes des machines corrompues, mais pas celles des machines fiables. L'Adversaire dispose d'une puissance de calcul permettant d'exécuter tout algorithme polynomial (mais, par exemple, pas de calculer une clef secrète à partir d'une clef publique). L'adversaire ne peut pas interférer avec les messages émis par les machines fiables.

6.2.2 L'algorithme, en résumé

Chaque machine dispose d'un compteur γ_i, initialisé à 0, et d'une variable $b_i \in \{0, 1\}$, initialisée à son vote initial. À chaque phase, lors du pas de communication, chaque machine adresse un unique message à chaque autre machine. Une machine peut adresser le message 0^* (resp. 1^*) pour signifier que dorénavant, elle n'enverra plus de messages mais les autres machines doivent considérer qu'elle reçoivent '0' (resp. '1') de sa part à chaque phase.

On suppose qu'un mot binaire R aléatoire a été déterminé initialement, aléatoirement et indépendamment des clefs publiques des participants. Chaque machine connaît ce mot R. On note $s_i(x)$ la signature du message x par la machine (de clef publique) i. On note $SIG_i(x)$ le mot binaire encodant l'information complète de signature $(i, x, s_i(x))$.

On rappelle que le nombre de machines est $n = 3t+1$. L'algorithme BBA* est une boucle comportant 3 phases (les $\bullet$), chacune subdivisée en un pas de communication et 3 pas de calcul. On note $m_i^r(j)$ le message reçu à la phase $r \in \{1, 2, 3\}$ par la machine i de la machine j. Pour toute valeur v on note $\#_i^r(v)$ le nombre de machines desquelles la machine i a reçu le message v à la phase $r \in \{1, 2, 3\}$.

Voici l'algorithme du point de vue de la machine i :

- Diffuser b_i à toutes les machines (soi compris).
  a. Si $\#_i^1(0) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 0$, émettre 0^* à tous, choisir 0 et terminer.
  b. Sinon, si $\#_i^1(1) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 1$.
  c. Sinon $b_i \stackrel{\text{def}}{=} 0$.
- Diffuser b_i à toutes les machines (soi compris).
  d. Si $\#_i^2(0) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 0$.
  e. Sinon, si $\#_i^2(1) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 1$, émettre 1^* à tous, choisir 1 et terminer.
  f. Sinon $b_i \stackrel{\text{def}}{=} 1$.
- Diffuser b_i et $SIG_i(R \cdot \gamma_i)$ à toutes les machines (soi compris).
  g. Si $\#_i^3(0) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 0$.
  h. Sinon, si $\#_i^3(1) \geq 2t+1$ alors $b_i \stackrel{\text{def}}{=} 1$.
  i. Sinon, $S_i \stackrel{\text{def}}{=} \{j : m_i^3(j) = SIG_j(R \cdot \gamma_i)\}$, $c_i \stackrel{\text{def}}{=} LSB(\min_{j \in S_i} H(SIG_j(R \cdot \gamma_i)))$, $b_i \stackrel{\text{def}}{=} c_i$, incrémenter γ_i et recommencer à la phase 1.

Cet algorithme donne la garantie suivante :

Théorème 6. *L'algorithme BBA* résout* Consensus Byzantin *avec probabilité* 1 *pour* $3t+1$ *machines en tolérant* t *pannes Byzantines.*

Précisons que la probabilité à laquelle il est fait référence ici a trait au comportement pseudo-aléatoire des primitives cryptographiques (H et SIG_i) et au choix du mot de référence R. Nous allons éclaircir ceci, et prouver le théorème, en exercices.

6.2.3 Exercices

On dit que les machines fiables sont *d'accord* si elles ont toutes la même valeur b_i. On suppose dans tous les exercices ci-après qu'il y a au plus t machines Byzantines parmi les $n = 3t+1$ machines.

Exercice 6 ★ On suppose que les machines fiables sont d'accord en début de phase 1. Peut-on prédire le déroulement de l'algorithme du point de vue d'une machine fiable quoi que fassent les machines Byzantines ? Expliquez.

Exercice 7 ★ Montrez que si les machines fiables sont d'accord au début d'une phase, alors elles choisissent toute cette valeur et terminent en au plus une exécution de chaque phase.

Exercice 8 ★★ Examinons le potentiel de nuisance de l'adversaire en considérant une exécution de l'algorithme sur un système à $n = 4$ machines dont $t = 1$ Byzantine.

a. Décrivez des votes initiaux et une exécution des deux premières phases au cours de laquelle les trois machines fiables ne réussissent pas à se mettre d'accord.
b. À quelle condition est-ce que l'adversaire peut maintenir les valeurs b_i des machines fiables différentes au cours de la troisième phase ?
c. Proposez une modélisation et une analyse probabilistes de cette condition.

Exercice 9 ★★ Intéressons nous à la phase 3. Supposons qu'au début de la phase 3 aucune machine n'ait encore terminé. Toutes les probabilités sont calculées au sens de la modélisation de l'exercice précédent.

a. Est-il possible que la condition de (g) soit vraie pour une machine fiable *et* que la condition de (h) soit vraie pour une autre ?

b. Intéressons-nous aux machines fiables pour lesquelles ni la condition de (g) ni celle de (h) n'est vraie. Donnez une minoration de la probabilité que toutes ces machines fiables aient la même valeur b_i après exécution du pas (i).

c. Donnez une minoration de la probabilité que toutes les machines fiables aient la même valeur b_i en sortie de phase 3.

Exercice 10 ★★ Voyons le dernier ingrédient de la preuve du théorème 6. On sait déjà, depuis l'exercice 7, que si toutes les machines fiables ont le même vote initial alors elles terminent toutes en choisissant ce vote. On sait aussi que tant qu'aucune machine fiable ne termine, chaque exécution des trois phases a une probabilité $\geq \alpha$ de les mettre d'accord, où α est une valeur raisonnable calculée à l'exercice 9. Il nous reste donc à prendre en compte le fait que des machines fiables peuvent terminer.

a. Montrez que si une des machines fiables termine à (a), alors toutes les machines fiables sont d'accord en sortie de cette phase 1.

b. Montrez que toute machine fiable termine avec probabilité 1.

c. Montrez que toutes les machines fiables qui terminent font le même choix.

Un dernier bon exercice (facultatif) consiste à synthétiser les idées des exercices ci-dessus en une preuve du Théorème 6.

6.3 Références bibliographiques

[3] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv :1607.01341*, 2016.

[4] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[5] Silvio Micali. Byzantine agreement, made trivial, 2018.

Notes personnelles

Annexes

Annexe A

Quelques rappels d'arithmétique

Toute donnée informatique (texte, image, base de données, ...) peut s'écrire comme *un* mot binaire, et est donc interprétable comme le nombre entier dont ce mot est l'écriture binaire. Quelques rappels de calcul en nombres entiers peuvent être utiles.

Un entier a divise un entier b s'il existe un entier k tel que $a = kb$. Pour tous entiers a, b, on note $\texttt{pgcd}(a, b)$ le plus grand diviseur commun à a et b ; si $\texttt{pgcd}(a, b) = 1$ on dit que a et b sont *premiers entre eux*. L'*identité de Bézout* énonce que pour tous entiers a, b il existe des entiers u, v tels que

$$au + bv = \texttt{pgcd}(a, b).$$

Étant donnés a et b, on peut calculer u et v par une modification assez simple de l'*algorithme d'Euclide*.

On note $a \mod b$ le reste de la division entière de a par b. Étant fixé un entier n, on note $\mathbb{Z}/n\mathbb{Z}$ l'ensemble $\{0, 1, \ldots, n-1\}$ muni des opérations suivantes :

$$a + b \stackrel{\text{def}}{=} (a + b) \mod n \qquad \text{et} \qquad a \cdot b \stackrel{\text{def}}{=} (ab) \mod n.$$

On peut vérifier que c'est un anneau commutatif. Remarquons cependant que 0 peut y être divisible : $3 \cdot 2 = 0 \mod 6$ par exemple. Les diviseurs de 0 n'admettent pas d'inverse, donc $\mathbb{Z}/n\mathbb{Z}$ n'est pas un corps en général.

Un entier p est *premier* si ses seuls diviseurs sont 1 et lui-même. Fixons p premier et prenons un entier $1 \leq q \leq p-1$. Les entiers p et q sont premiers entre eux, aussi l'identité de Bézout assure qu'il existe des entiers u et v tels que $qu + pv = 1$, et ainsi $q \cdot u = 1 \mod p$. Cela assure que tout élément non nul de $\mathbb{Z}/p\mathbb{Z}$ a un *inverse* ; cet inverse peut se calculer au moyen de l'algorithme d'Euclide.

La *taille* d'un entier est définie comme le nombre de chiffres de sa plus petite écriture binaire. Ainsi, l'entier cinq s'écrit 101 mais aussi 00101 ou encore 0000101 ; sa taille est 3. Plus généralement, tout entier $n \geq 1$ est de taille $\approx \log_2 n$ puisque $2^{\lfloor \log_2 n \rfloor} \leq n \leq 2^{\lceil \log_2 n \rceil}$.

Clarifions la complexité du calcul de certaines opérations arithmétiques, qui s'exprime comme une fonction de la taille de ses entrées :

- On connait un algorithme de complexité polynomiale pour tester si un entier donné est premier (autrement dit, ce problème est dans P).
- On ne connaît pas d'algorithme de complexité polynomiale pour factoriser un entier non premier donné. On n'a pas de preuve qu'il n'existe pas d'algorithme polynomial pour ce problème.
- On ne connaît pas d'algorithme de complexité polynomiale pour calculer, étant donnés des entiers $x \leq p$, le logarithme discret de x dans $\mathbb{Z}/p\mathbb{Z}$. (Ici, la taille de l'entrée est le nombre de bits de p.) On n'a pas de preuve qu'il n'existe pas d'algorithme polynomial pour ce problème.

Annexe B

Formalisation de la notion de preuve sans divulgation de connaissance

La preuve de résolution de sudoku présentée au Chapitre 3 est un exemple de *preuve à divulgation nulle de connaissances* (*zero knowledge proof*). Voyons comment ces preuves se formalisent en théorie de la complexité.

Protocole interactif

Une *Machine de Turing interactive* est une machine de Turing qui dispose des rubans suivants : un ruban d'entrée (lecture seule), un ruban de travail, deux rubans de communication (un en lecture, un en écriture), et un ruban servant de source de bits aléatoires.

Un *protocole interactif* est une paire (P, V) de machines de Turing interactives qui partagent le même ruban d'entrée, et telles que le ruban de communication en lecture de l'une est le ruban de communication en écriture de l'autre. On appelle P le *prouveur* et V le *vérificateur*.

Dans un protocole interactif, les machines P et V sont actives alternativement, P passant la main à V dès écriture d'un message sur le ruban $r_{P\to V}$ (et inversement). À tout moment, P ou V peut interrompre le calcul, et lorsque V est actif il peut décider de terminer le calcul en acceptant ou en refusant l'entrée. Le temps de calcul pris par (P, V) sur une entrée x est le nombre total de pas de calcul effectués par le vérificateur V au cours de toutes ses phases d'activité.

Système de preuve interactif à divulgation nulle de connaissances

Dans la définition de NP, on adosse la décision d'accepter une entrée x à l'examen d'un certificat y. L'idée d'un système de preuve interactif est de séparer les choses : P connaît y et doit convaincre V d'accepter x. Cette connaissance de y par P n'est pas modélisée explicitement, mais remarquons que le temps de calcul pris par P n'est pas comptabilisé. On peut donc imaginer que P commence par tester préalablement *tous* les certificats possibles y jusqu'à en trouver un qui permette de faire accepter x.

On s'intéresse à un protocole interactif (P, V) pour un problème de décision (E, A) qui, quand il termine sur une entrée x, le fait après un nombre de pas de calcul de V polynomial en $|x|$. On mesure la qualité de (P, V) par les critères suivants. Les probabilités sont ici prises relativement à l'aléa des sources de bits aléatoires des machines de Turing interactives concernées.

- La *complétude* de (P, V) est la probabilité que P réussisse à convaincre V d'accepter x lorsque $x \in A$. On souhaite qu'elle soit forte. Formellement, on souhaite que pour tout k, pour tout $x \in A$ assez grand, la probabilité que (P, V) termine en temps polynomial en $|x|$ et accepte x est au moins $1 - |x|^{-k}$.
- La *fiabilité* de (P, V) est la probabilité qu'un prouveur P' (pas nécessairement P) réussisse à convaincre V d'accepter x lorsque $x \notin A$. On souhaite qu'elle soit faible. Formellement, on souhaite que pour tout k, pour tout $x \in E \setminus A$ assez grand, pour tout protocole interactif (P', V), la probabilité que (P', V) termine et accepte x est au plus $|x|^{-k}$.

- La *divulgation* de (P, V) est la quantité d'information qu'un vérificateur V' (pas nécessairement V) peut déduire de son intéraction avec P dans un protocole (P, V'). Cela se formalise en termes statistiques : le fonctionnement des machines de Turing étant aléatoire (puisqu'elles ont accès à une source de bits aléatoires), il s'agit de savoir si les distribution des données sur les rubans de V' s'écarte substantiellement de toutes les distributions que V' peut calculer en temps polynomial en $|x|$. Nous ne formaliserons pas plus avant ; pour approfondir, on pourra consulter :

  Goldwasser, Shafi, Silvio Micali, and Charles Rackoff. "The knowledge complexity of interactive proof systems." SIAM Journal on computing 18.1 (1989) : 186-208.

Annexe C

Principes d'une analyse de cycle de vie

Une *analyse de cycle de vie* (ACV en abrégé, LCA en anglais pour *life cycle assessment*) établit un bilan quantitatif et multicritère des impacts d'un système (produit, entreprise, ...) sur l'environnement. C'est un procédé normalisé (série ISO 14000). Nous donnons ici quelques clefs utiles à la lecture d'une ACV, et renvoyons, pour une introduction plus détaillée et de qualité, à

https://fr.wikipedia.org/wiki/Analyse_du_cycle_de_vie

Article scientifique VS article de presse

Soulignons que contrairement à un article de presse, un article scientifique doit, pour être publié, avoir été validé au travers d'un examen contradictoire indépendant et, généralement, dans une temporalité sereine. Dans le cas des analyses d'impact environnemental, cette exigence est aussi valable pour les publications venant d'industriels. Cela donne à une telle source un caractère *opposable*.

Unité fonctionnelle

Une ACV commence par la définition précise de l'objet d'étude. Cet objet est ramené à une quantité de référence. Ainsi, l'ACV de la fabrication d'un tissu doit spécifier sur quelle quantité de tissu l'étude porte : est-ce 1 m^2, 1 m^3, 1 kg, ... ? Bien que les résultats soient convertibles, ce choix a une importance dès lors que l'on utilise l'ACV pour *comparer* deux manières de remplir une fonction. Le choix de l'unité de référence est donc guidé par la *fonction* que l'on souhaite voir le système remplir. Dans notre exemple textile, si le tissu sert à fabriquer des bâches l'unité de surface sera adaptée ; s'il sert à rembourrer des coussins, le volume est plus pertinent. Etc.

Étapes

Une ACV comporte trois étapes distinctes : l'*inventaire* des matières et énergies mis en jeu par le système, la *traduction* de cet inventaire en impact environnemental, et l'*interprétation* de ces résultats.

Pour établir l'inventaire, il convient de modéliser l'ensemble des flux de matière et d'énergie mis en jeux dans la production d'une unité fonctionnelle, de la collecte des matières premières au traitement des déchets finaux, en passant par les différents usages. Le bilan est qualitatif et quantitatif.

La traduction de l'inventaire suppose de préciser la question posée. Soulignons que s'il est possible de comparer, par exemple, les contributions à l'effet de serre du CO_2 et du méthane, cette comparaison varie dans le temps, ne serait-ce qu'en raison de durées différentes de séjour dans l'atmosphère. Ainsi, si 1 unité de méthane a le même impact que ~ 80 unités de CO_2 à horizon 20 ans, cette quantité tombe à ~ 30 à horizon 100 ans. Les modèles utilisés pour ces traductions sont donc importants, et ils sont, sans surprise, normalisés.

Type d'analyse

Il existe deux principaux modèles d'ACV. L'ACV **attributionnelle** (ACV-A) examine un système installé, supposé en régime permanent et ne supposant pas de remise en cause des chaines d'approvisionnement. L'objectif de l'ACV-A est de déterminer la part de l'impact environnemental de l'existant qu'il

convient d'attribuer au système étudié. Par exemple, en ACV-A, la traduction de l'énergie consommée par le système en impact environnemental peut se baser sur le mix énergétique local de la région.

L'ACV **conséquentielle** (AVC-C) examine un système dont la mise en place perturberait les chaines d'approvisionnement auquel il s'adosserait, et vise à mesurer les conséquences de cette installation. Par exemple, en ACV-C, la traduction de l'énergie consommée par le système en impact environnemental ne se basera sur le mix énergétique local de la région que pour le surplus d'énergie disponible, et mesurera en totalité les nouvelles sources d'énergie qu'il faudra développer. L'analyse conséquentielle opère à base de scénarii.

Annexe D

Barrières théoriques en calcul distribué

Tenir une chaîne de blocs suppose de résoudre des problèmes d'élection et de consensus avec régularité, équité et fiabilité. Les solutions mises en œuvre par Bitcoin, à base de preuve de travail et de tolérance d'erreur, ne donnent aucune garantie formelle et présentent de gros inconvénients (coût énergétique insoutenable et long délai pour assurer l'irréversibilité d'une transaction). Le chapitre 6 discutera d'autres méthodes en calcul distribué, mais avant cela, il convient d'examiner quelques résultats d'impossibilité. À l'exception d'un rapide échauffement, on se concentre dorénavant sur le problème du consensus.

Les objectifs sont que vous...

- sachiez adapter des exemples simples d'impossibilité en calcul distribué,
- compreniez les principes et la portée d'une preuve d'impossibilité en calcul distribué.

D.1 Échauffement : élection déterministe sans identifiant unique

Revenons, pour cette section seulement, au problème de l'élection dans un système distribué à réseau :

Élection (sans panne)

Entrée : Rien

Sortie : Chaque machine choisit 0 ou 1. Exactement une machine choisit 1.

On a vu que lorsque chaque machine dispose d'un identifiant unique, une approche simple consiste à faire circuler ces identifiants sur le réseau et à déclarer élue la machine d'identifiant maximal (ou, selon les préférence à la conception de l'algorihtme, minimal, médian, ...).

Quand on ne dispose pas de manière de casser la symmétrie entre les machines, il s'avère qu'il est **impossible** d'élire un meneur. On illustre d'abord l'idée de la preuve sur un cas simple, avant de la prolonger en exercice :

Proposition 7. *Il n'existe pas d'algorithme déterministe qui résout* Élection (sans panne) *dans le modèle synchrone à réseau complet pour deux machines indistinguables.*

Idée de la preuve. On va raisonner par contradiction : on suppose qu'il existe un algorithme qui réalise la tâche décrite, et on démontre que cet algorithme échoue. Chaque machine n'a qu'un seul canal de communication sortant et un seul canal de communication entrant. Examinons les premières phases de calcul et de communication :

- Les deux machines commencent la phase 1 dans le même état.
- Donc les deux machines émettent le même premier message m_1 (éventuellement vide), sur leur seul canal sortant.

- Donc les deux machines reçoivent le même premier message (c'est toujours m_1).
- Donc les deux machines commencent la phase 2 dans le même état.

Le même raisonnement permet de montrer par récurrence sur k que (i) les deux machines commencent la phase k dans le même état, (ii) les deux machines effectuent le même calcul à la phase k, (iii) les deux machines émettent le même message au cours de la phase k, et (iv) les deux machines reçoivent le même message au cours de la phase k. Chaque machine termine son calcul après l'exécution d'un nombre fini de phases, et cela doit se passer à la même phase pour les deux machines à cause de (ii). De plus, les deux machines doivent faire le même choix final, et ce n'est pas une réponse correcte au problème de l'élection. □

Il est facile de formaliser complètement cette preuve une fois un modèle de calcul fixé (RAM, machine de Turing, ...). Ce modèle est naturellement le même pour chaque machine, et permet d'expliciter ce qu'est un algorithme, un état interne et un pas de calcul. La preuve ne requiert que deux propriétés du modèle : (i) qu'il commence l'exécution d'un programme donné dans le même état interne, et (ii) que l'état interne après un pas de cacul dépende uniquement de l'état interne et des messages reçus au début de ce pas de calcul. Ces deux propriétés sont ce que l'on désigne par l'adjectif « déterministe » dans l'énoncé de la Proposition 7.

Une autre manière de comprendre l'idée de preuve de la Proposition 7 est que le système se comporte *comme si* chaque machine était "en boucle", c'est à dire recevait ses propres messages. Ce système modifié est composé de 2 machines identiques, sans interaction extérieure, qui exécutent le même programme. Si une machine fait un choix final, les deux machines font *ce* choix final, et le problème de l'élection n'est pas résolu.

D.1.1 Exercices

Exercice 1 ★★ Adaptez cette preuve pour expliquer pourquoi aucun algorithme ne peut résoudre le problème de l'élection dans le modèle synchrone, sans panne, à réseau **circulaire** de n machines indistinguables.

Exercice 2 ★★★ Quel énoncé de théorie des graphes permettrait d'étendre l'idée de la preuve de la Proposition 7 au cas d'un système synchrone sans panne de n machines formant un réseau complet ? Que savez-vous dire de la validité de cet énoncé ?

D.2 Consensus synchrone avec pannes Byzantines

Intéressons-nous maintenant à la gestion des comportements malveillants.

D.2.1 Le modèle

On travaille ici dans un modèle synchrone à n machines et à réseau complet. Les machines disposent d'un identifiant unique à valeur dans $[n]$, c'est à dire qu'elles sont numérotées de 1 à n. Rappelons qu'une machine dans un système distribué a un *comportement Byzantin* si elle ne se comporte pas conformément à l'algorithme. Cela peut être dû à un problème technique (défaillance d'un des ports de communication par exemple) ou à un comportement malicieux (prise de contrôle de la machine et modification du programme par un tiers malveillant). Lorsque l'on modélise un système où des comportements Byzantins sont possibles, on dit qu'une machine est *fiable* si elle se comporte conformément à l'algorithme.

D.2.2 La difficulté de démasquer les machines Byzantines

Face à la possibilité de pannes Byzantines, nous allons résister à la tentation, naturelle, de chercher à identifier les machines fautives. Dans certaines situations cela s'avère impossible ! Pour expliquer cela, plongeons-nous dans la métaphore poliorcétique utilisée originellement pour ce problème.

Une armée fait le siège d'une ville. L'armée est dirigée par un général et organisée en divisions, chacune dirigée par un colonel. Les colonels peuvent communiquer entre eux et avec le général par l'intermédiaire de messagers (les communications ne sont pas cryptographiquement signées). Parmi le général et les colonels, certains sont déloyaux, c'est à dire qu'ils sont passés à l'ennemi ; leur objectif est de faire attaquer une partie (et une partie seulement) des divisions dirigées par des colonels loyaux.

Le général envoie un ordre ("attaquer" ou "se retirer") à chacun de ses colonels. Chaque colonel doit décider si sa division attaque ou se retire. Les colonels loyaux doivent résoudre, de manière distribuée, le problème suivant :

a. Les colonels loyaux doivent prendre la même décision. En effet, si seule une partie d'entre eux attaque ils vont se faire tailler en pièces.

b. Si le général est loyal, alors les colonels loyaux doivent obéir à son ordre. On note qu'un général loyal donnera le même ordre à tous ses colonels.

Chaque colonel doit prendre sa décision seul, mais peut communiquer avec les autres colonels. On souhaite un algorithme distribué permettant d'atteindre cet objectif.

Revenons à la question de démasquer les participants déloyaux. Supposons que le 1er colonel reçoive les messages suivants :

- Un ordre d'attaquer de la part du général.
- Un message du 2ème colonel l'informant que le général lui a donné l'ordre de se retirer.

Il y a manifestement de la déloyauté... mais elle peut venir du général ou du 2ème colonel et le 1er colonel n'a aucun moyen de le déterminer.

La métaphore militaire motive l'étude des pannes Byzantines par la résistance à la malveillance. Soulignons que ces pannes incluent aussi des dysfonctionnements "chaotiques" au cours desquels des machines se mettent à transmettre des messages incohérents aux autres machines du système. L'article de Driscoll et coll. [6] en donne quelques exemples réels.

D.2.3 Bien poser le problème

Le problème de Consensus Byzantin ne demande pas de forcer une machine Byzantine à se soumettre au consensus, mais vise à former un consensus entre les machines fiables, malgré les interférences des machines Byzantines. Formellement :

Consensus Byzantin

Entrée : Chaque machine a un vote initial valant 0 ou 1

Sortie : Chaque machine choisit 0 ou 1. Ces choix doivent satisfaire les conditions suivantes :

a. Toutes les machines fiables font le même choix.

b. Si toutes les machines fiables ont le même vote initial x, leur choix final est x.

Le problème esquissé en Section D.2.2 est un peu différent et se formalise ainsi :

RELIABLE BROADCAST

Entrée : Une machine distinguée (émettrice) a un vote initial $x \in \{0,1\}$.

Sortie : Chaque machine non-émettrice choisit 0 ou 1. Ces choix doivent satisfaire les conditions suivantes :

a. Toutes les machines fiables font le même choix.

b. Si la machine émettrice est fiable, alors les machines fiables choisissent x.

Soulignons que ce problème est atypique au sens l'on conçoit un programme particulier pour la machine émettrice. Une troisième variante, généralement étudiée conjointement avec ces problèmes, consiste à mettre d'accord les machines fiables sur l'ensembles de leurs votes initiaux :

COHÉRENCE INTERACTIVE

Entrée : Chaque machine a un vote initial valant 0 ou 1

Sortie : Chaque machine choisit un vecteur dans $\{0,1\}^n$. Ces choix doivent satisfaire les conditions suivantes :

a. Toutes les machines fiables font le même choix v.

b. Si la $i$ème machine est fiable, alors v_i égale son vote initial.

Soulignons que pour résoudre ces problèmes, il n'est pas nécessaire de déterminer *quelles* sont les machines non fiables : on souhaite juste qu'elles ne puissent pas empêcher les machines fiables de se mettre d'accord.

Un algorithme pour l'un des problèmes ci-dessus **tolère b pannes Byzantines** si *toute* exécution au cours de laquelle au plus b machines ont un comportement Byzantin aboutit à une solution.

D.2.4 Principe d'un RELIABLE BROADCAST tolérant des pannes Byzantines

Commençons par montrer qu'il est possible de résoudre l'un de ces problèmes, disons RELIABLE BROADCAST, en présence de pannes Byzantines. L'algorithme de la machine émettrice est élémentaire :

```
emettre(tous, vote initial)
terminer
```

Bien évidemment, si la machine émettrice n'est pas fiable mais Byzantine, elle peut exécuter le code qu'elle veut...

Dans ce qui suit, on définit l'*élément majoritaire* d'une liste de n éléments comme (i) soit l'unique élément présent strictement plus de $n/2$ fois, (ii) soit 0 si aucun élément n'est présent strictement plus de $n/2$ fois.

Tolérance à 1 panne Byzantine

L'algorithme suivant tolère 1 panne dès lors que $n \geq 4$.

```
1 synchro
2 C[id] = message reçu de l'émettrice, 0 si rien n'a été reçu
3 emettre(tous,C[id]) puis synchro puis recevoir
4 pour i=1..n et différent de id
5    C[i] = valeur reçue de la machine i, 0 si rien n'a été reçu
6 w = valeur majoritaire dans C[1..n]
7 choisir(w) et terminer
```

Cela se prouve par une analyse de cas.

Cas 1 : la machine émettrice est fiable. Notons x son vote initial. Alors pour chaque machine non-émettrice fiable, le vecteur C contient au moins $n-2$ copies de x (voire $n-1$ selon le comportement de la machine Byzantine). On a $n-2 > \frac{n-1}{2}$ dès lors que $n \geq 4$. Toutes les machines fiables feront donc le même choix et ce choix coïncide avec le vote initial de la machine émettrice.

Cas 2 : la machine émettrice est Byzantine. Dans ce cas, les machines non-émettrices sont toutes fiables. Elles ont donc toutes le même vecteur C et font donc le même choix (la valeur majoritaire de C, quelle qu'elle soit).

D.2.5 Quelques exercices

Exercice 3 ★ Supposons que l'on ait 3 colonels et qu'il y ait au plus un participant (général ou colonel) déloyal. Est-ce possible de le démasquer ? Même question si on a 99 colonels et un participant déloyal.

Exercice 4 ★★ Supposons que l'on tente de résoudre Consensus Byzantin en deux phases : (i) chaque machine adresse son vote initial à toutes les autres, (ii) chaque machine choisit le vote majoritaire (0 en cas d'égalité). À combien de pannes Byzantines cet algorithme résiste-t-il ?

Exercice 5 ★★ Examinons les liens entre les trois problèmes introduits en Section D.2.3 :

a. Supposons que l'on dispose d'un algorithme résolvant Cohérence interactive. Proposez une solution à Consensus Byzantin. (La première ligne de votre solution sera `v := choix résultat de la cohérence interactive`.) À combien de pannes Byzantine cette solution résiste-t-elle ?

b. Supposons que l'on dispose d'un algorithme résolvant Reliable broadcast. Proposez une solution à Cohérence interactive. À combien de pannes Byzantine cette solution résiste-t-elle ?

c. Supposons que l'on dispose d'un algorithme résolvant Consensus Byzantin. Proposez une solution à Reliable broadcast. À combien de pannes Byzantine cette solution résiste-t-elle ?

Exercice 6 ★★ Décrivez une exécution de l'algorithme de la Section D.2.4 pour $n = 3$ où une panne Byzantine réussit à faire échouer le *reliable broadcast*.

Exercice 7 ★★★ **(optionnel)** Proposez un algorithme qui résout *reliable broadcast* et tolère 2 pannes Byzantines si n est assez grand. Pour quelle valeur de n cette tolérance est-elle valide ?

Exercice 8 ★★★ **(optionnel)** Pour chacune des réductions données à l'exercice 5, donnez « l'efficacité » de l'algorithme construit à partir de « l'efficacité » de l'algorithme supposé, pour les mesures suivantes d'efficacité :

a. le nombre de pannes Byzantines tolérées,

b. le nombre de phases,

c. le nombre de messages transmis.

D.2.6 Il n'existe pas d'algorithme synchrone supportant $\lceil n/3 \rceil$ pannes Byzantines

Il s'avère qu'aucun algorithme ne peut résoudre Cohérence interactive pour n machines en présence de $\lceil n/3 \rceil$ pannes Byzantines. Pour établir que *pour tout* algorithme distribué, *il existe* un *scénario d'exécution* dans lequel la cohérence interactive échoue, il nous faut avant tout formaliser les concepts d'algorithme et de scénario dans ce contexte.

On note $M = \{1, 2, \ldots, n\}$ l'ensemble des machines, V l'ensemble des valeurs qu'elles peuvent se communiquer ($V = \mathbb{N}$ par définition, mais le résultat restera valide pour $V = \{0, 1\}$), et M^* l'ensemble des mots sur l'alphabet M. Rappelons que mM^* est l'ensemble des mots sur l'alphabet M qui commencent par m.

Un *scénario* est une application de M^* dans V. Un *m-scénario* est une application de mM^* dans V. Si α est un scénario, on note $\alpha_{|m}$ le m-scénario obtenu en restreignant α à mM^*. L'ensemble M^* modélise les transmissions de messages avec intermédiaires : le mot $m_1m_2m_3m_4$ désigne une communication de m_4 à m_1 passant de m_4 à m_3, puis de m_3 à m_2, puis de m_2 à m_1. Un scénario décrit le message parvenant au destinataire lors de chaque transmission avec intermédiaires. Par convention, dans un scénario α, pour $m \in M$ on interprète $\alpha(m)$ comme la valeur initiale que reçoit la machine m.

Soit $F \subseteq M$ un sous-ensemble de machines. On dit qu'un scénario α est *cohérent* avec F si

$$\forall q \in F, \forall m \in M, \forall w \in M^*, \quad \alpha\left(pqw\right) = \alpha\left(qw\right),$$

c'est-à-dire que les machines de F relaient les messages sans les modifier. Une *m-exécution* est une application qui associe une valeur de V à toute paire (α_m, m') où α_m est un m-scénario et $m' \in M$. Intuitivement, une m-exécution E_m associe à (α_m, m') la valeur $E_m(\alpha_m, m')$ que m attribue à m' dans son vecteur final de choix si l'ensemble des communications qui parviennent à m est décrit par α_m.

Une famille $\{E_m : m \in M\}$ d'exécutions *résout la cohérence interactive avec b pannes Byzantines* si pour tout $F \subseteq M$ avec $|F| \geq m - b$, et pour tout scénario α cohérent avec F,

$$\forall p, q \in F, \qquad E_p(\alpha_{|p}, q) = \alpha(q). \tag{D.1}$$

$$\forall p, q \in F, \forall r \in M, \quad E_p(\alpha_{|p}, r) = E_q(\alpha_{|q}, r) \tag{D.2}$$

Autrement dit, si p et q sont des machines fiables (dans F), alors la valeur que p décide pour q égale la valeur initiale de q, et les valeurs que p et q décident pour toute machine r (fiable ou pas, donc dans M) sont égales.

Nous pouvons enfin énoncer précisément le théorème d'impossibilité, dû à Lamport, Pease et Shostak [8].

Théorème 8. *Pour $|V| \geq 2$ et $|M| \leq 3b$, il n'existe pas de famille $\{E_m : m \in M\}$ qui résout la cohérence interactive avec b pannes Byzantines.*

Faute de temps, nous n'en présentons la preuve (pour $|M| = 3$ et $b = 1$) qu'en Annexe E.

D.3 En asynchrone une panne est ingérable

Intéressons-nous maintenant à la gestion des simples pannes (extinction), mais dans un modèle asynchrone.

D.3.1 Le modèle et le problème

On travaille ici dans un modèle asynchrone à n machines et à réseau complet. Les machines disposent d'un identifiant unique à valeur dans $[n]$, c'est à dire qu'elles sont numérotées de 1 à n. Rappelons qu'une *panne* désigne le fait qu'une des machines du système *arrête* de fonctionner : elle ne reçoit plus les messages, n'effectue plus aucun calcul interne et n'émet plus de messages. Une panne est un problème moins général qu'un comportement Byzantin mais suffit déjà à modéliser la possibilité de plantage, de coupure d'électricité, d'incendie, ... Une machine qui n'est pas en panne est appelée *fonctionnelle*. Comme vu au Chapitre 4, le problème de consensus avec pannes se formule comme suit :

CONSENSUS TOLÉRANT AUX PANNES

Entrée : Chaque machine a un vote initial valant 0 ou 1.

Sortie : Chaque machine choisit un vote final valant 0 ou 1. Ces votes doivent satisfaire les conditions de cohérence globale suivantes :

a. Toutes les machines fonctionnelles ont le même vote final.

b. Le vote final égale l'un des votes initiaux.

Notons bien que l'on ne demande pas à ce que le vote final égale le vote initiale d'une machine fonctionnelle.

D.3.2 Le théorème de Fischer, Lynch et Paterson

Nous avons étudié en Section 4.4 un algorithme qui résout CONSENSUS TOLÉRANT AUX PANNES dans un modèle synchrone en tolérant un nombre f de pannes fixé a priori. Pour le modèle asynchrone il s'avère que cet algorithme échoue. Plus généralement, Fischer, Lynch et Paterson [7] ont établi dans les années 1980 qu'aucun algorithme ne peut résoudre ce problème.

Théorème 9 (Théorème FLP)**.** *Il n'existe pas d'algorithme asynchrone résolvant* CONSENSUS TOLÉRANT AUX PANNES *et tolérant une faute.*

La présentation (et la preuve) du Théorème 8 a commencé par une modélisation des communication au sein d'un système distribué. La preuve du Théorèm 9 commence elle aussi par une étape de modélisation, mais portant ici sur les *ordonnancements* possibles des pas de calcul. Notre objectif ici est de saisir les idées principales de la preuve et, dans une certaine mesure, leur portée. Pour une preuve complète, on renvoie à la source originale [7].

D.3.3 Graphe d'états du système

On suppose défini la notion d'*état* d'une machine à un instant donné comme un ensemble d'informations suffisant à décrire complètement sa situation interne. Par exemple, dans le cas d'une machine de Turing interactive, il s'agit de l'ensemble des règles de transition, de l'état courant dans l'automate, de la position des têtes de lecture/écriture sur les différents rubans et du contenu de ces rubans s'ils ne sont pas aléatoires. La *configuration* de notre système distribué à un instant t^* est la paire (e, m) où $e = (e_1, e_2, \ldots, e_n)$ avec e_i l'état de la $i$ème machine à l'instant t^* et m est l'ensemble des messages en transit sur le réseau (c'est à dire envoyés mais pas encore reçus).

Une *exécution* d'un algorithme sur un système distribué asynchrone correspond à une suite de configurations du système qui correspond à certaines règles. En bref, si on note (e, m) et (e', m') deux configurations successives, on doit avoir :

- il existe $1 \leq i \leq n$ tel que $e_j = e'_j$ pour $j \neq i$, et
- il existe un événement ϕ s'appliquant à e_i tel que $\phi(e_i) = e'_i$, et
- l'événement ϕ est cohérent avec le changement entre m et m'.

Un *événement* est un pas élémentaire de calcul du système Σ. Cela peut être la réception d'un message, le changement d'état, ou l'émission d'un message par une machine ; on dit que l'événement *porte* sur la machine réalisant l'action en question. On dit qu'un événement ϕ est *applicable* à la configuration (e, m) si ϕ peut se produire lorsque le système est en configuration (e, m). (Par exemple, l'événement `réception du message M par la machine 1` suppose que $M \in m$.) Si un événement ϕ est applicable à (e, m), on note $\phi(e, m)$ la configuration dans laquelle le système se trouve immédiatement après ϕ s'il était en (e, m) immédiatement avant.

Il est pratique de définir un graphe orienté $\mathcal{G}$ dont les sommets sont les configurations possibles du système distribué, et dont les arêtes sont les paires $((e, m), (e', m'))$ telles qu'il existe un événement ϕ avec $\phi(e, m) = (e', m')$. On dit alors qu'une configuration (e', m') est *accessible* depuis (e, m) s'il existe un chemin de (e, m) à (e', m') dans $\mathcal{G}$.

D.3.4 Premier pas de preuve, en exercices

Exercice 9 ★ On considère l'algorithme LCR vu à la section **??**, qui résout le problème de l'élection dans un anneau asynchrone de machines à identifiant unique

```
1  emettre(droite, id)
2  record = id
3  m = recevoir(gauche)
4  si m > record:
5      record = m
6      emettre(droite, record)
7  si m == id:
8      emettre(droite, "STOP")
9      choisir(1) et terminer
10 si m == "STOP":
11     emettre(droite, "STOP")
12     choisir(0) et terminer
13 retourner à la ligne 3
```

a. Proposez une description de l'état d'une machine qui suffit à décrire l'exécution de cet algorithme.
b. Décrivez la configuration initiale du système.
c. Décrivez l'ensemble des événements pouvant être le premier événement d'une exécution.
d. Quel est l'ordre de grandeur du nombre de sommets du graphe $\mathcal{G}$ induit par la notion de configuration que vous avez proposé.

Exercice 10 ★★ Soit (e, m) l'état d'un système distribué Σ. Soient ϕ_i et ϕ_j deux événements qui s'appliquent à (e, m) et s'appliquent à des machines différentes. Montrer que $\phi_i \circ \phi_j(e, m) = \phi_j \circ \phi_i(e, m)$.

Exercice 11 ★★ On va maintenant mettre en place la base de la preuve du théorème FLP. Supposons, par l'absurde, qu'il existe un algorithme $\mathcal{A}$ qui résout Consensus tolérant aux pannes.

a. On suppose fixées les notions d'état d'une machine et de configuration du système. Combien existe-t-il de configurations initiales du système ?

Deux configurations initiales sont *voisines* si elles diffèrent dans l'état d'exactement une machine. On définit un graphe auxiliaire $\mathcal{H}$ (distinct de $\mathcal{G}$) comme suit : les sommets de $\mathcal{H}$ sont les configurations initiales et les arêtes de $\mathcal{H}$ sont les paires de configurations initiales voisines.

b. Dessinez $\mathcal{H}$ pour $n = 2$ et $n = 3$ machines.
c. Décrivez le graphe $\mathcal{H}$ pour n quelconque.

Une configuration de Σ est *finale* si elle est accessible depuis au moins une configuration initiale et si toutes les machines y ont émis leur vote final. Dans une configuration finale, toutes les machines ont le même vote final puisque l'algorithme résout le consensus ; la configuration finale *vaut* 0 ou 1 selon la valeur de ce vote final. Une configuration (e, m) est 0-*valente* si toutes les configurations finales accessibles depuis elle valent 0. On définit de même les configurations 1-*valentes*.

d. Décrivez une configuration initiale 0-valente et une configuration initiale 1-valente.

Une configuration (e, m) est *bivalente* si elle n'est ni 0-valente, ni 1-valente.

e. Démontrez que si aucune configuration initiale n'est bivalente, alors il existe deux configurations initiales voisines, l'une 0-valente et l'autre 1-valente.

D.3.5 Une panne peut provoquer de la bivalence, qui se propage

Continuons l'analyse amorcée à l'exercice 11. Lorsqu'une panne survient sur la $i$ème machine, cela a trois effets :

- L'ensemble des configurations change : on supprime des configurations l'état de la $i$ème machine. On transforme la configuration courante au moment de la panne en supprimant l'état de la $i$ème machine.
- Les configurations qui étaient non-finales car seul le vote final de la $i$ème machine manquait sont désormais considérées comme finales.
- On ne peut plus appliquer au système aucun événement s'appliquant à la $i$ème machine. (On peut toujours, en revanche, faire arriver des messages envoyés par la $i$ème machine avant sa panne.)

Cela conduit naturellement à réviser les notions de 0-/1-/bivalence. La première idée clef de la preuve de FLP est qu'une panne suffit à produire une configuration bivalente.

Proposition 10. *Pour tout algorithme, le système a une configuration initiale bivalente, éventuellement après une panne.*

Idée de preuve. Supposons que le système n'a pas de configuration initiale bivalente (sans quoi l'énoncé est trivialement vrai). Comme on l'a vu à l'exercice 11e, il existe alors deux configurations initiales voisines, I et J, qui sont respectivement 0-valente et 1-valente. Puisqu'elles sont voisines, I et J diffèrent par l'état d'exactement une machine, disons i. Une panne de la $i$ème machine tombe en panne dès le début de l'algorithme, provoque l'identification des configurations initiales I et J en une nouvelle configuration initiale qui s'avère bivalente. □

La seconde idée clef de la preuve de FLP est que la bivalence peut se perpétuer :

Proposition 11. *Pour toute configuration bivalente (e, m), il existe toujours une autre configuration bivalente (e', m') accessible depuis (e, m).*

Le théorème 9 découle immédiatement des propositions 10 et 11 : pour tout algorithme, il existe une entrée bivalente (quitte à supposer une panne bien choisie) et une suite d'événements qui maintient le système dans une configuration bivalente et l'empêche donc d'atteindre une configuration finale.

La preuve de la proposition 11 est un peu technique. Fixons ϕ un événement qui peut s'appliquer à (e, m). Notons C l'ensemble des configurations accessibles depuis (e, m) *sans* appliquer ϕ. Le modèle asynchrone autorise à ce que tout message soit retardé arbitrairement longtemps. Cela assure que ϕ est applicable à toute configuration de C. Notons $D \stackrel{\text{def}}{=} \{\phi(e', m') \colon (e', m') \in C\}$.

Lemme 12. *D contient une configuration bivalente.*

Idée de preuve. On raisonne par l'absurde. Supposons que D ne contienne pas de configuration bivalente.

On commence par montrer que D contient à la fois des configurations 0-valentes et des configurations 1-valentes. Il existe deux configurations E_0 et E_1, respectivement 0-valente et 1-valente, accessibles depuis (e, m). Si $E_i \in C$ alors on pose $F_i \stackrel{\text{def}}{=} e(E_i) \in D$. Sinon, E_i est obtenu en appliquant ϕ et est donc soit dans D, auquel cas on pose $F_i \stackrel{\text{def}}{=} E_i$, soit accessible depuis une configuration $F_i \in D$. Dans les deux cas, F_i appartient à D et est i-valente (dans le second cas, la i-valence découle de l'hypothèse que D n'a pas de configuration bivalente).

On prouve ensuite que C contient deux configurations C_0, C_1 telles que (i) $\phi(C_i)$ est i-valente pour $i = 0, 1$ et (ii) on peut passer d'un C_i à l'autre par un seul événement ϕ'. Par symmétrie des rôles, supposons que $\phi'(C_0) = C_1$.

Si ϕ et ϕ' portent sur des machines différentes, elles commutent. On a alors que

$$D_1 = \phi(C_1) = \phi(\phi'(C_0)) = \phi \circ \phi'(C_0) = \phi' \circ \phi(C_0) = \phi'(D_0),$$

ce qui est impossible car appliquer un événement à une configuration 0-valente ne peut pas produire une configuration 1-valente. Si ϕ et ϕ' portent sur la même machine, une analyse similaire (mais un peu plus délicate) permet de conclure. □

Le lemme 12 suffit à établir la proposition 11 puisque la configuration bivalente contenue dans D est accessible depuis (e, m) en au moins une étape de calcul.

D.3.6 Un théorème c'est bien, une preuve c'est encore mieux

La preuve du théorème FLP repose sur deux idées. D'une part, une panne suffit à produire de la bivalence. D'autre part, si un état initial est bivalent alors il existe une séquence finie non vide de pas de calcul qui mène à un autre état bivalent. On peut en tirer diverses conséquences **faciles** mais qui **ne découlent pas du seul énoncé** du théorème, par exemple :

- Le théorème FLP n'exclue pas l'existence d'un algorithme pour le consensus qui, en présence de pannes, *est correct quand il termine*, mais risque de ne jamais terminer. C'est par exemple le cas de Paxos.
- Si un algorithme est correct sans panne et termine en temps fini, alors le résultat *ne peut pas* dépendre de l'exécution. En effet, il faudrait pour cela qu'un état initial soit bivalent, et d'après la proposition 11, il existerait alors une exécution au cours de laquelle l'algorithme ne termine pas.

D'où mon mot de la fin :

> On gagne beaucoup à ne pas accepter un théorème comme un fait, mais à comprendre au moins l'idée générale de sa preuve.

Remarquons que comprendre l'idée d'une preuve peut se faire en laissant de côté certains détails techniques. Ici, ne pas tout comprendre dans la preuve du lemme 12 ne nous a pas gêné.

D.4 Conclusion : retour sur Bitcoin

Revenons sur le système Bitcoin à la lumière de ce que l'on a vu à cette séance. Tout d'abord, le système Bitcoin étant ouvert, il faut supposer quand on l'analyse que certaines machines peuvent être Byzantines. Notons que ce n'est pas, par exemple, le cas du système Libra imaginé par Facebook.

Bitcoin suppose de résoudre un problème d'élection pour déterminer quelle machine écrit la page suivante de la blockchaine. La « solution » consiste à déclarer élue la première machine à proposer une solution au problème de minage. Cette « solution » est formellement incorrecte, puisque deux machines peuvent proposer une solution à un intervalle de temps trop court pour que le système ne les départage.

Bitcoin suppose aussi de résoudre un problème de consensus pour décider si une nouvelle page proposée par un mineur est acceptée. La « solution » déployée consiste à garder en mémoire toutes les pages cryptographiquement et historiquement correctes, et à considérer comme acceptées les pages formant le chemin le plus long. À nouveau, cette « solution » est formellement incorrecte, puisqu'une page considérée à un moment comme acceptée peut être ultérieurement rejetée (puis à nouveau acceptée, etc.)

Ces imperfections ne chagrinent pas que les théoriciens amateurs d'algorithmes propres : elles conduisent à l'annulation de transactions et à de l'insécurité. À cela s'additionne le problème, majeur, du coût énergétique du minage. C'est donc un défi important des blockchaines que de déployer de meilleures solutions. Les résultats d'impossibilité que l'on a prouvé ou évoqué établissent une certitude : aucune solution ne peut être parfaite.

D.5 Références bibliographiques

[6] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.

[7] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2) :374–382, 1985.

[8] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2) :228–234, 1980.

Annexe E

Preuve du théorème d'impossibilité de Lamport, Pease et Shostak

Rappelons le théorème d'impossibilité, dû à Lamport, Pease et Shostak [8].

Théorème. Pour $|V| \geq 2$ et $|M| \leq 3b$, il n'existe pas de famille $\{E_m : m \in M\}$ qui résout la cohérence interactive avec b pannes Byzantines.

Cette annexe en donne une preuve pour $|M| = 3$ et $b = 1$, avec quelques exercices optionnels pour tester votre compréhension. Notons $M = \{a, b, c\}$ et notons v, v' deux éléments distincts de V.

Trois scénarii

On définit trois scénarii α, β et γ comme suit.

a. Pour tout mot $w \in M^*$ qui ne termine pas par c, on pose

$$\alpha(w) = \beta(w) = \gamma(w) \stackrel{\text{def}}{=} v.$$

b. Pour $w \in M^*c$, on définit $\alpha(w)$, $\beta(w)$ et $\gamma(w)$ récursivement.

(a) On initialise la récursion en définissant directement les scenarii sur les mots de longueur 1 et 2 :

$$\begin{gathered}\alpha(c) = \alpha(ac) = \alpha(bc) = \alpha(cc) \stackrel{\text{def}}{=} v \\ \beta(c) = \beta(ac) = \beta(bc) = \beta(cc) \stackrel{\text{def}}{=} v' \\ \gamma(c) = \gamma(ac) = \gamma(cc) \stackrel{\text{def}}{=} v \quad \text{et} \quad \gamma(bc) \stackrel{\text{def}}{=} v'\end{gathered}$$

(b) Pour tout $p \in M$ et $w \in M^*c$, on pose

$$\begin{array}{lll}\alpha(paw) \stackrel{\text{def}}{=} \alpha(aw), & \alpha(pbw) \stackrel{\text{def}}{=} \beta(bw), & \alpha(pcw) \stackrel{\text{def}}{=} \alpha(cw) \\ \beta(paw) \stackrel{\text{def}}{=} \alpha(aw), & \beta(pbw) \stackrel{\text{def}}{=} \beta(bw), & \beta(pcw) \stackrel{\text{def}}{=} \beta(cw) \\ \gamma(paw) \stackrel{\text{def}}{=} \gamma(aw), & \gamma(pbw) \stackrel{\text{def}}{=} \gamma(bw), & \gamma(acw) \stackrel{\text{def}}{=} \alpha(cw) \quad \text{et} \quad \gamma(bcw) \stackrel{\text{def}}{=} \beta(cw)\end{array}$$

Exercice 12 ★ Calculez :

a. $\alpha(bac)$, $\beta(bac)$ et $\gamma(bac)$.

b. $\alpha(abc)$, $\beta(abc)$ et $\gamma(abc)$.

c. $\alpha(acb)$, $\beta(acb)$ et $\gamma(acb)$.

d. $\alpha(acc)$, $\beta(acc)$ et $\gamma(acc)$.

Propriétés des scénarii

Les définitions assurent que $\gamma_{|a} = \alpha_{|a}$ et $\gamma_{|b} = \beta_{|b}$. Autrement dit, γ se comporte comme α "du point de vue" de a et comme β "du point de vue" de b.

Exercice 13 ★★ Prouvez que $\gamma_{|a} = \alpha_{|a}$ et $\gamma_{|b} = \beta_{|b}$, c'est-à-dire que

$$\forall w \in M^*, \qquad \gamma(aw) = \alpha(aw) \quad \text{et} \quad \gamma(bw) = \beta(bw).$$

On pourra procéder par récurrence sur la longueur du mot w.

Cohérence

Rappelons qu'un scénario δ est cohérent avec $F \subseteq M$ si

$$\forall q \in F, \forall p \in M, \forall w \in M^*, \quad \alpha\left(pqw\right) = \alpha\left(qw\right).$$

Exercice 14 ★ Vérifiez que α est cohérent avec $\{a, c\}$, que β est cohérent avec $\{b, c\}$ et que γ est cohérent avec $\{a, b\}$.

Aboutissement

Supposons que $\{E_a, E_b, E_c\}$ résout la cohérence interactive avec 1 pannes Byzantines. On a donc, par définition de "résoudre...", que pour tout $F \subseteq M$ avec $|F| \geq 2$, et tout scénario δ cohérent avec F,

$$\forall p, q \in F, \qquad E_p(\delta_{|p}, q) = \delta(q). \tag{E.1}$$

$$\forall p, q \in F, \forall r \in M, \quad E_p(\delta_{|p}, r) = E_q(\delta_{|q}, r) \tag{E.2}$$

(Ici, on répète simplement la définition.) Comme α est cohérent avec $\{a, c\}$, pour $p = a$ et $q = c$ la condition (E.1) donne

$$E_a(\alpha_{|a}, c) = \alpha(c) = v.$$

Comme β est cohérent avec $\{b, c\}$, pour $p = b$ et $q = c$ la condition (E.1) donne

$$E_b(\beta_{|b}, c) = \beta(c) = v'.$$

Puisque γ est cohérent avec $\{a, b\}$, pour $p = a$, $q = b$ et $r = c$ la condition (E.2) donne

$$E_a(\gamma_{|a}, c) = E_b(\gamma_{|b}, c).$$

Comme $\alpha_{|a} = \gamma_{|a}$ et $\beta_{|b} = \gamma_{|b}$, on obtient

$$v = E_a(\alpha_{|a}, c) = E_a(\gamma_{|a}, c) = E_b(\gamma_{|b}, c) = E_b(\beta_{|b}, c) = v'$$

et donc $v = v'$, une contradiction.