blème particulier à chaque demande. Mais la première méthode permet une prise en compte plus fine de la surcharge du service à l'instant de la demande ou d'informations connues de S concernant D.

Le délai nécessaire pour que le demandeur résolve le problème posé est le plus souvent fondé sur un calcul à faire par le demandeur. Le temps qu'il met à résoudre le problème dépend donc de sa puissance de calcul et les machines puissantes sont avantagées. Aussi a-t-on envisagé des preuves de travail exigeant beaucoup de mémoires (plutôt que beaucoup de calculs) ou nécessitant la recherche d'informations sur le réseau. Dans ce dernier cas, les machines puissantes ne sont pas avantagées, puisque ce qui nécessite du temps est lié à la vitesse de circulation des informations sur le réseau, qui est la même pour tout le monde.

L'idée des preuves de travail est due à Cynthia Dwork et Moni Naor qui, dès 1993, suggérèrent cette méthode pour lutter contre les spams. Le système *Hashcash* conçu par Adam Back en 1997 (indépendamment de C. Dwork et M. Naor) et servant aujourd'hui de base dans la plupart des mises en œuvre pratiques des preuves de travail, utilise des « fonctions à sens unique » pour définir les problèmes soumis et en contrôler finement la difficulté *(voir l'encadré 2 pour des exemples)*.

Une fonction f à sens unique se calcule facilement, mais s'inverse difficilement. Autrement dit, il est rapide de passer de x à $f(x) = y$, mais difficile de trouver, à partir d'un y donné, un x tel que $y = f(x)$. On exigera souvent aussi que les valeurs de $f(x)$ quand x varie se présentent comme si elles étaient tirées au hasard.

Fonctions à sens unique, avec ou sans paramètre

Si l'on dispose d'une telle fonction f, on en déduit facilement une preuve de travail : le service S choisit un y et exige du demandeur D qu'il trouve un x tel que $f(x) = y$. Le demandeur ne peut guère faire mieux qu'essayer des valeurs de x, jusqu'à trouver la bonne, ce qui est impossible à faire rapidement. Il s'agit ici d'une impossibilité pratique : trouver le x est possible en essayant de manière ordonnée toutes les x envisageables, mais cela nécessite du temps.

Plus subtil, car cela ne demande pas l'intervention de S pour la formulation du problème (donc utilisable pour les preuves de travail sans échange), on utilise une fonction à sens unique dépendant d'un paramètre p, $f_p(x)$, et dont le résultat, y, peut être interprété comme un nombre réel positif. Une telle interprétation est toujours possible : on écrit y en binaire, par exemple 01001110, et on assimile cette suite de 0 et de 1 au nombre réel dont l'écriture en base 2 est 0,01001110. Le service S attend que D lui propose un x tel que $f_p(x) < y$, où p est par exemple le message que D veut envoyer et y un nombre réel, appelé seuil, fixé à l'avance par S. La valeur de y détermine très précisément la difficulté de la preuve de travail, car plus y est petit, plus il est difficile de trouver un x convenable.

Les preuves de travail fournissent aussi un moyen simple d'organiser des courses entre machines sur un réseau. Or ces courses sont au cœur des protocoles de fonctionnement des cryptomonnaies.

Ces monnaies, dont le *bitcoin* est l'exemple le plus connu, fonctionnent sans autorité centrale, la gestion des comptes étant opérée par les machines des volontaires qui se contrôlent mutuellement. Ces contrôles empêchent toute manipulation du cahier de compte de la monnaie (la « blockchain »), lequel indique qui détient des devises *(voir Pour la Science, décembre 2013)*.

1. L'idée des preuves de travail

Pour limiter le pouvoir de nuisance que chacun possède du fait de la puissance de son ordinateur, on peut soumettre des énigmes obligeant les machines à de longs calculs et dont seul l'aboutissement leur donne le droit de mener une action, comme envoyer un message électronique ou accéder à un service en ligne. Une machine soumise à ces énigmes ne pourra pas envoyer des milliers de messages non sollicités (spams), ou participer à une tentative de mise en panne d'un service informatique en le submergeant de requêtes (attaque par déni de service).

Le temps de calcul imposé est un prix à payer pour que l'ordinateur acquière le droit de mener une action. Il doit travailler et le prouver par un résultat. La solution produite est la preuve qu'il a fourni ce travail. C'est l'idée des « preuves de travail ».

Les problèmes qu'on demande de résoudre pour ces défis doivent être asymétriques. Il faut que le calcul de la solution prenne du temps, mais que la vérification soit rapide ; sinon, celui qui se protégerait par les preuves de travail serait autant ennuyé que ceux dont il tenterait de se protéger.

On connaît une multitude de problèmes asymétriques. En voici deux exemples :

– L'entier b étant fixé, et p étant un grand nombre premier, trouver un nombre a tel que $a^2 = b \bmod p$. Ce problème de la « racine carrée modulo un grand nombre premier » a été proposé comme preuve de travail dès 1993 par Cynthia Dwork et Moni Naor.

– Trouver les deux nombres premiers p et q dont le produit est le nombre n qu'on vous a donné sans vous avoir dévoilé p ni q. Il est facile de formuler de telles énigmes – on multiplie deux grands nombres premiers entre eux, ce qui produit n – mais aucun algorithme de factorisation rapide n'est connu, et il faut donc du temps pour retrouver p et q à celui qui ne connaît que n. Il ne faut pas prendre p et q trop grands, car le problème de la factorisation de n deviendrait impossible à résoudre en un temps raisonnable.

Un troisième exemple – le plus important, car le plus utilisé – est expliqué en détail dans les encadrés 2 et 3 et se fonde sur les fonctions « à sens unique ».

Jean-Michel Thiriet

Pour qu'il y ait un intérêt pratique à participer à ce contrôle collectif sans lequel la cryptomonnaie n'existe plus, une récompense est donnée à intervalles réguliers à l'une des machines tenant les comptes. Aujourd'hui par exemple, 25 *bitcoins* sont distribués toutes les dix minutes par le protocole *Bitcoin*. Cette machine gagnante est désignée par un procédé fondé sur une preuve de travail dont le fonctionnement doit être parfait pour que le protocole résiste aux attaques. C'est la plus belle application à ce jour de l'idée des preuves de travail, et de nombreuses caractéristiques de *Bitcoin* en dépendent. Donnons quelques précisions.

Chaque machine candidate essaie de résoudre un problème dont les paramètres sont fixés par le protocole *Bitcoin* et qui est formulé sans intervention d'un service centralisé. Il s'agit d'un problème du type « trouver un x tel que $f_p(x) < y$ » évoqué plus haut. La fonction f à sens unique est connue de tous, et c'est la fonction de hachage SHA-256. On l'utilise beaucoup en cryptographie et les spécialistes s'accordent à considérer qu'elle a les propriétés voulues pour servir dans une preuve de travail. La première machine qui résout le problème gagne la compétition et emporte les 25 *bitcoins*. Le protocole ajuste régulièrement le seuil donné par le nombre y, qui fixe la difficulté de la preuve de travail en prenant en compte les concours précédents, de telle façon que le temps moyen entre deux concours soit de dix minutes environ.

Des preuves de travail pour les bitcoins

Cette émission contrôlée, prévisible et faible de *bitcoins* toutes les dix minutes est la seule façon d'en créer. Cela a pour intérêt que les détenteurs de *bitcoins* sont certains de ne pas être victimes des phénomènes inflationnistes provoqués par l'émission massive de devises, comme c'est parfois le cas pour les monnaies contrôlées par les banques centrales.

Participer à ce concours porte le nom de « minage », car le travail de calcul fait par les machines évoque celui d'un mineur dans une mine d'or, qui le conduit à s'enrichir s'il a la chance de trouver une pépite. Les informaticiens propriétaires des machines participant à ce concours permanent, répété toutes les dix minutes, se nomment entre eux des « mineurs ».

Il est important de comprendre que puisqu'il n'y a pas d'autorité centrale gérant les *bitcoins* (il en va de même pour les autres cryptomonnaies), l'acceptation de la preuve de travail se fait collectivement par tous les mineurs. Quand l'un d'eux a réussi à relever le défi de trouver un x tel que $f_p(x) < y$, il envoie un message sur le réseau, qui prévient les autres mineurs. Ceux-ci vérifient alors, chacun de son côté, que la solution proposée est correcte et se remettent au travail pour résoudre le défi suivant.

Le temps nécessaire pour qu'un gagnant soit trouvé n'est pas fixé avec précision : la

2. Le hachage

Une fonction de hachage associe, à un x donné (une suite de caractères ou un nombre) de longueur quelconque une valeur $f(x)$ de taille fixe nommée « empreinte de x » ou « condensé de x » ou *hash*. Le passage de x à $f(x)$ se fait par une série d'opérations qui commencent en général en découpant x en petits morceaux (d'où le nom de hachage), qui sont ensuite rassemblés et emmêlés de façon à obtenir un mélange aussi parfait que possible. Pour la fonction classique de hachage SHA-256, la longueur de $f(x)$ est de 256 bits ; il y a donc $2^{256} = 1{,}157 \times 10^{77}$ valeurs possibles, ce qui est considérable.

Les fonctions de hachage à usage cryptographique ont les propriétés suivantes.

(a) Elles sont « à sens unique » : le passage de x à $f(x)$ se calcule facilement et rapidement, mais pour un y donné, trouver un x tel que $y = f(x)$ est impossible en pratique.

(b) Il est impossible en pratique de trouver deux valeurs x et x' différentes telles que $f(x) = f(x')$.

(c) Elles produisent des valeurs assimilables à des valeurs aléatoires uniformes (prises dans l'ensemble des valeurs possibles) et, en particulier, même si x et x' sont deux suites de caractères n'ayant qu'un seul caractère différent, les valeurs $f(x)$ et $f(x')$ n'ont aucun lien apparent.

Voici un exemple. La donnée x est transformée en une suite de 0 et de 1 qui est découpée en morceaux de longueur 8. Les morceaux sont ensuite additionnés comme des nombres en base 2. On ne garde du résultat que les quatre chiffres en positions 3, 4, 5 et 6 à partir de la droite :

x=01010010100101010010010101 0 → 01010010 + 10010101 + 00100101 + 010 = 10000111 0 → $f(x)$ = 0011.

Cet exemple est bien sûr trop simple pour avoir les propriétés *(a)*, *(b)* et *(c)*. La fonction de hachage SHA-256 mélange soigneusement les bits de la donnée x selon un algorithme public (voir http://en.wikipedia.org/wiki/SHA-2).

Les fonctions de hachage cryptographiques ont de multiples usages dont voici trois exemples.

– Elles permettent d'assurer l'intégrité d'un gros fichier x, auquel on lui associera son empreinte $f(x)$, beaucoup plus courte. Celui qui téléchargera x s'assurera que x lui parvient sans erreur et en totalité en calculant l'empreinte du fichier qu'il aura reçu, qui doit être la même que celle publiée sur le site. Si x a mal été copié, il le saura, car la probabilité que la modification n'ait pas changé l'empreinte est infinitésimale.

– Au lieu de mémoriser les mots de passe d'accès aux comptes sur un service en ligne, celui-ci ne mémorise que les empreintes des mots de passe. Le serveur peut vérifier que les utilisateurs donnent les bons mots de passe, mais si un pirate accède à sa liste d'empreintes, il ne pourra rien en faire.

– Pour produire une suite de nombres aléatoires dépendant d'un fichier x, on calculera par exemple $f(x.1)$, $f(x.2)$,... où $x.i$ désigne le fichier x suivi de l'écriture du nombre i (par exemple en binaire). En changeant le x, on a autant de suites de nombres aléatoires qu'on veut (ce qui permet par exemple de chiffrer des messages).

L'importance des fonctions de hachage est telle, en cryptographie, qu'un grand soin est mis à les élaborer et à les soumettre à toutes sortes de tests et d'attaques. Le NIST, aux États-Unis, propose des fonctions bien contrôlées. La fonction SHA-256 utilisée par *Bitcoin* en est une. Le NIST publie aussi des recommandations concernant le choix et l'usage des fonctions de hachage. Il a récemment organisé une compétition pour concevoir une nouvelle génération de fonctions de hachage qui a abouti à la norme SHA-3 en 2012. Voir : http://www.nist.gov/itl/csd/ct/hash_competition.cfm.