Université de Montréal

SIMULATION DE PROTOCOLES DE CRYPTOGRAPHIE QUANTIQUE

par

Jean-François Riendeau

Département d'Informatique et de Recherche Opérationnelle
Faculté des Arts et des Sciences

Mémoire présenté à la Faculté des Études Supérieures
en vue de l'obtention du grade de Maître ès Sciences (M.Sc.)
en Informatique

1^{er} Décembre 1995

Université de Montréal

Faculté des Études Supérieures

Ce mémoire intitulé

SIMULATION DE PROTOCOLES DE CRYPTOGRAPHIE QUANTIQUE

présenté par

Jean-François Riendeau

a été evalué par un jury composé des personnes suivantes :

Pierre McKenzie

(président-rapporteur)

Claude Crépeau

(directeur de recherche)

Gilles Brassard

(co-directeur)

Geňa Hahn

(membre du jury)

Mémoire accepté le :

19 Octobre 1995

SOMMAIRE

Depuis l'invention de la cryptographie quantique en 1970 par Wiesner, de nombreuses équipes de recherche de par le monde travaillent à construire l'appareil quantique, dont le premier prototype a été réalisé par Bennett et al. Nous faisons référence, entre autres, aux travaux d'une équipe de British Telecom [27] et d'une équipe de l'Université de Genève [12]. La réalisation physique de ces appareils est encore expérimentale et coûteuse. C'est pourquoi nous proposons une solution temporaire: la simulation de protocoles de cryptographie quantique, à l'aide d'un logiciel de simulation. Toutefois, ce logiciel a été pensé pour pouvoir facilement intégrer un prototype réel du canal quantique.

Le développement de ce logiciel est une partie importante de ce mémoire. Il a été conçu pour rendre plus facile la simulation de protocoles cryptographiques, en fournissant un grand nombre de structures de données utilisées fréquemment dans la définition formelle de ces protocoles comme par exemple les vecteurs de bits, les corps finis, les polynômes et les codes correcteurs d'erreurs. De plus, il permet la manipulation de particules virtuelles, en utilisant les principes de la physique quantique, tels les opérateurs unitaires et les mesures de Von Neumann.

Dans un premier temps, nous présenterons les détails de ce monde quantique simulé. Nous verrons, à l'aide d'exemples, comment sont implantées ces particules virtuelles. Nous décrirons les deux outils pouvant faire interagir les particules et les mécanismes sous-jacents. Nous présenterons le canal quantique en tant que primitive cryptographique, ainsi que les autres primitives quantiques disponibles aux usagers.

Par la suite, nous décrirons les structures de données prédéfinies, leur utilité et certaines de leurs fonctionnalités. Nous présenterons aussi les quatre protocoles prédéfinis, en mentionnant leur historique, leur utilité et un pseudo-code décrivant la façon dont ils ont été implantés. En annexe, nous fournirons une description de certains protocoles alternatifs, ainsi que les sources en C++ des protocoles prédéfinis.

REMERCIEMENTS

Tout d'abord, je tiens à remercier Claude Crépeau, pour m'avoir d'abord initié au merveilleux monde de la cryptographie lors du cours IFT-6180 d'automne 1992. Je tiens aussi à le remercier pour son support tout au long de ma maîtrise.

Je tiens aussi à remercier Gilles Brassard, pour l'inspiration qu'il m'a procurée. Ses commentaires furent incisifs, mais très appréciés.

Un merci tout-à-fait spécial à William K. Wootters. Le séminaire qu'il a présenté à l'Université de Montréal sur les mesures quantiques fut une base sur laquelle le chapitre 2 s'est fondé. De plus, il a su répondre avec patience à mes 1001 questions tordues sur la physique quantique.

Je tiens finalement à remercier le CRSNG pour son appui financier, sans lequel la réalisation de cette maîtrise n'aurait pas été possible.

TABLE DES MATIÈRES

LISTE DES FIGURES

INTRODUCTION

1. La cryptographie et l'informatique

À l'origine, la cryptographie était la science du chiffrement des messages. Elle décrivait les façons de transformer un message sensé en un message chiffré, dont le sens (s'il en a encore un) n'a aucun rapport avec le message original, de sorte que seul le destinataire légitime soit en mesure de retrouver le message original. Les premiers cryptosystèmes (systèmes de chiffrement) reposaient sur une information secrète que partageaient l'envoyeur et le destinataire. Nous appelons cette information la clé du cryptosystème. Pour certains cryptosystèmes, cette clé est très simple, comme par exemple, la substitution d'une lettre de l'alphabet par la lettre précédente. Dans d'autres cas, le calcul du message chiffré est un processus pénible pour un cerveau humain. On crée alors des machines conçues pour chiffrer automatiquement des messages. ENIGMA, utilisée par les Allemands pendant la Seconde Guerre Mondiale, en est un exemple.

Vers 1975, la cryptographie connaît un renouveau: les cryptosystèmes à clés publiques. Dans ces systèmes, une clé publique sert au chiffrement des messages vers un destinataire A, et une clé secrète (connue de A seul) permet le déchiffrement du message. Un des premiers parmi ces systèmes, RSA, est basé sur la théorie des nombres. Le chiffrement d'un message passe par l'exponentiation d'un grand nombre associé au message, modulo un produit de 2 grands nombres premiers. Nous avons ici un autre exemple de calculs difficiles à accomplir par un simple être humain.

Ce très bref aperçu montre que la cryptographie et l'informatique *flirtent* ensemble depuis longtemps[1]. C'est encore plus vrai depuis l'avènement des protocoles cryptographiques. Un protocole cryptographique peut être défini formellement comme étant un groupe de machines de Turing pouvant communiquer entre elles. Les méthodes de communication varient selon le modèle théorique utilisé (diffusion, présence de bruit, tripatouillage, etc...). La plupart

[1] D'ailleurs, à l'Université de Montréal, la cryptographie est une branche de recherche du DIRO.

des protocoles cryptographiques publiés sont exprimés sous forme d'algorithmes, exprimés la plupart du temps dans un mélange de pseudo-code et de langage courant, chaque algorithme décrivant le comportement d'un des participants au protocole. Voici un exemple de protocole cryptographique très simple:

1. *Alice* choisit un bit b au hasard.
2. *Bob* choisit un bit b' au hasard.
3. *Bob* prend l'annuaire du téléphone et choisit une page p au hasard de même parité que son bit b'.
4. *Bob* choisit au hasard un numéro de téléphone présent sur cette page et l'annonce à *Alice*.
5. *Alice* annonce la valeur de son bit b à *Bob*.
6. *Bob* annonce le numéro de la page p et son bit b' à *Alice*.
7. *Alice* vérifie que le numéro de téléphone choisi par *Bob* est bien présent sur la page. S'il n'y est pas, *Alice* rejette et boude *Bob*...
8. S'il y est, alors si $b = b'$, on retourne PILE, sinon on retourne FACE.

Ce protocole, très simple en apparence, permet à 2 personnes de jouer à PILE ou FACE par téléphone (à la condition d'avoir accès au même annuaire), sans donner d'avantage à un des participants. La sécurité du protocole repose sur le fait qu'il est difficile de:

- trouver la page à partir du numéro de téléphone seulement (Attaque par Alice).
- trouver 2 pages de parités différentes partageant un même numéro de téléphone, en excluant bien sûr les fins et débuts de page (Attaque par Bob) [2].

De plus, la difficulté de ces attaques augmente avec la taille de l'annuaire. Dans tous les protocoles cryptographiques, nous retrouvons cette notion de paramètre dont dépend la sécurité.

2. La cryptographie et la physique quantique

En plus de dépendre d'un paramètre de sécurité, la sécurité d'un protocole repose souvent sur une hypothèse calculatoire. Dans le cas du protocole décrit plus haut, l'hypothèse calculatoire est qu'un être humain passerait beaucoup de temps à parcourir l'annuaire (et serait rapidement blasé), alors que le protocole, exécuté de façon honnête, ne dure pas longtemps. La sécurité du protocole serait anéantie si l'annuaire était accédé via une base de données à accès très rapide. Les hypothèses calculatoires les plus utilisées sont la factorisation (par un

[2] Cette attaque est aujourd'hui beaucoup plus facile... En effet, il arrive souvent que 2 personnes, dont le nom de famille est différent, partagent le même domicile (et le même numéro de téléphone), que ce soit dans un couple marié dont la femme a conservé son nom de jeune fille, ou des conjoints de fait.

ordinateur *classique*[3]), le logarithme discret et certains problèmes NP-complets. Or, on ne sait toujours pas s'il existe ou non un algorithme efficace pour résoudre ces problèmes. Si c'était le cas, la sécurité de bien des protocoles serait compromise.

La cryptographie quantique quant à elle ne nécessite aucune hypothèse calculatoire. La seule hypothèse est bien sûr que la physique quantique soit correcte. Le premier protocole de cryptographie quantique fut imaginé par Wiesner en 1970 [29]. En 1984, Bennett et Brassard conçoivent un protocole d'échange de clés, paru dans [2], utilisant une primitive appelée *canal quantique*, utilisant la polarisation des photons pour transmettre une information aléatoire. Un premier prototype physique de ce protocole fut réalisé par Bennett, Bessette, Brassard, Salvail et Smolin. Depuis, plusieurs équipes travaillent sur différents aspects de la cryptographie quantique:

- Un meilleur prototype du canal quantique. Plusieurs équipes de recherche travaillent sur cet aspect, notamment une équipe de British Telecom [27] et une équipe suisse [12].
- Conception d'autres protocoles: [10] [5] [1].

La physique quantique, en libérant les protocoles de cryptographie quantique du besoin d'une hypothèse calculatoire, a cependant rendu l'écriture et la compréhension des preuves de sécurité de ces protocoles beaucoup plus difficiles. Les futurs spécialistes en cryptographie quantique devront être des cryptologues chevronnés et des physiciens accomplis.

3. Simulation logicielle de protocoles

Pourquoi voudrait-on simuler un protocole de cryptographie quantique? Les 2 précédentes sections devraient commencer à répondre à cette question. Tout d'abord, la définition d'un protocole en tant qu'algorithme suggère qu'une implantation permettrait de voir le protocole en action et ainsi de mieux le comprendre. Ceci est principalement un objectif didactique.

De plus, l'implantation physique d'un canal quantique est encore expérimentale, donc instable et coûteuse. D'où l'utilité d'un modèle mathématique simulant le comportement, au niveau quantique, de véritables particules. Le logiciel de simulation **QSim**, conçu et implanté dans le cadre de cette maîtrise, est donc un outil doublement intéressant. Je vous souhaite beaucoup de plaisir à l'utiliser, c'est-à-dire au moins autant que j'en ai eu à le concevoir...

[3] Shor a découvert un algorithme efficace pour la factorisation entière, de même que pour le logarithme discret fonctionnant sur l'ordinateur *quantique*. Voir [24].

CHAPITRE 1

PRELIMINAIRES

1. Notations

Dans ce mémoire, nous utiliserons plusieurs notations qui valent la peine d'être mentionnées:

- Le symbole $\oplus$ dénotera l'opération "ou-exclusif", soit entre 2 bits, soit entre 2 vecteurs de bits.
- Le symbole $\wedge$ dénotera l'opération "et", soit entre 2 bits, soit entre 2 vecteurs de bits.
- Le symbole $\odot$ dénotera le produit scalaire entre vecteurs de bits. Si $u = (u_0, \ldots, u_{n-1})$ et $v = (v_0, \ldots, v_{n-1})$, le produit scalaire entre les deux est calculé selon la formule suivante:

$$u \odot v = \bigoplus_{i=0}^{n-1} (u_i \wedge v_i)$$

- Le symbole $\otimes$ dénotera le produit tensoriel. Il peut s'agir d'un produit tensoriel entre 2 vecteurs (produisant un vecteur):

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}$$

  Dans ce cas, la longueur du vecteur produit est égale au produit des longueurs des vecteurs d'entrée. Par exemple, le produit tensoriel d'un vecteur de longueur m et d'un vecteur de longueur n sera un vecteur de longueur mn. Le produit tensoriel peut aussi s'appliquer entre 2 matrices (produisant une matrice):

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \otimes \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae & af & be & bf \\ ag & ah & bg & bh \\ ce & cf & de & df \\ cg & ch & dg & dh \end{bmatrix}$$

Dans ce cas, les dimension de la matrice produite seront les produits des dimensions des matrices en entrée. Par exemple, le produit tensoriel d'une matrice $m \times n$ et d'une matrice $p \times q$ sera une matrice $mp \times nq$. Par un léger abus de notation, il est aussi possible d'exprimer le produit tensoriel entre 2 ensembles. Cela veut dire en fait:

$$A \otimes B = \{a \otimes b \mid a \in A, b \in B\}$$

- Si A est une matrice, A^T dénotera la transposée de A.
- Si A est une matrice complexe, $A^\dagger$ dénotera la transposée conjuguée de A.
- La fonction d'entropie en base 2, notée H_2, est calculée selon la formule suivante:

$$H_2(p) = -p \log p - (1-p) \log(1-p)$$

2. Modèle de communication

Le logiciel suit le modèle Serveur-Clients. L'architecture de communication suit le diagramme suivant:

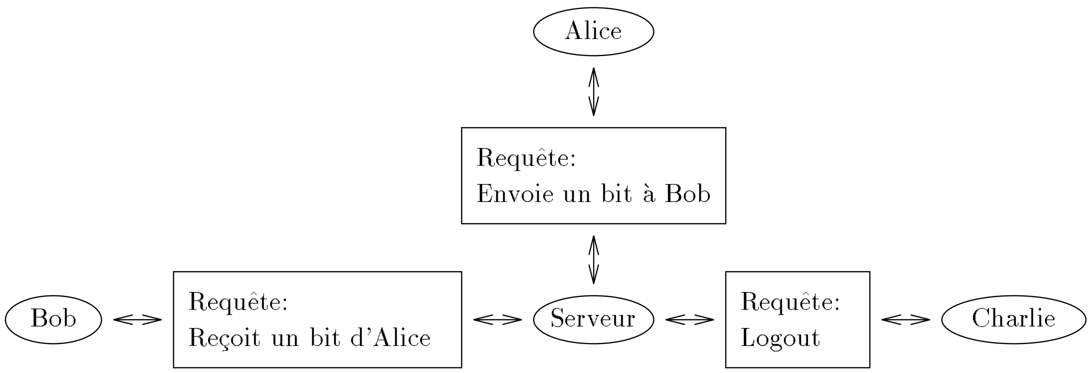

Figure 1.1. Modèle Serveur-Clients

Les connections utilisées pour relier les clients au serveur peuvent être locales ou de type Internet. Ces dernières permettent d'utiliser des serveurs non-locaux, et même de participer à un protocole avec un partenaire se trouvant outre-mer. Dans les 2 cas, un point de contact est utilisé par le serveur pour détecter les nouvelles connections. Un serveur, quelque soit le type de connection utilisé, commence par écouter le point de contact. Un client qui veut se connecter à ce serveur doit le lui faire savoir via ce point de contact. Une fois la connection établie, les 2 programmes communiquent en s'échangeant des requêtes. Typiquement, c'est le client qui envoie une requête au serveur et le serveur lui répond. Tout se passe de cette façon, du *login* jusqu'au *logout*, en passant par les transferts de données entre clients. Le serveur est comme une gare de triage qui interprète certains messages et aiguille les autres vers leurs destinations.

CHAPITRE 2

SIMULATION D'UN UNIVERS

1. Introduction

Pour simuler un canal quantique, il faut simuler les propriétés des photons au niveau quantique. Les principes utilisés pour réaliser cette simulation ont été glanés à gauche et à droite, et ne sont pas le fruit d'une formation intensive dans le domaine. Le but de ce chapitre est de présenter et d'expliquer ces principes. Les théoriciens de la physique quantique seront probablement horrifiés par ma terminologie et mon point de vue très opérationnel de leur science. J'estime néanmoins que ces explications, dans leur forme actuelle, seront d'une grande aide à ceux qui, comme moi, n'ont pas le génie de la physique quantique.

En premier lieu, nous présenterons certains fondements mathématiques de la physique quantique tels que les notions de vecteurs d'état, d'opérateurs unitaires et de mesures. Nous verrons comment ces notions ont été implantées dans **QSim** pour en faire des outils de simulation. Pour finir, nous présenterons notre contribution originale à ce mémoire: un algorithme de factorisation d'états.

2. Un modèle mathématique

2.1. Polarisation d'un photon. Considérez un faisceau de lumière. Selon le modèle ondulatoire de la lumière, ce faisceau est constitué d'une superposition d'ondes électromagnétiques. Isolez-en une. Cette onde est en fait 2 ondes, une électrique ($\vec{E}$), l'autre magnétique ($\vec{B}$), les 2 étant perpendiculaires entre elles et perpendiculaires à la direction de propagation de l'onde, comme le montre la Figure 2.1a. Si nous utilisons un plan de référence fixe passant par la direction de propagation, ce plan crée un angle avec le plan de l'onde électrique. Cet angle est la *polarisation* de l'onde. Et, à cause de la correspondance entre les modèles ondulatoires et corpusculaires de la lumière, nous pouvons parler de polarisation de photons. La Figure 2.1b illustre cet angle.

2.2. État et vecteur d'état. Tout d'abord, voici quelques définitions. À cet instant, nous dirons qu'un *état* est une description d'une ou de plusieurs propriétés d'une particule. Une *base* est définie comme étant un ensemble d'états parfaitement distinguables. Une base, en

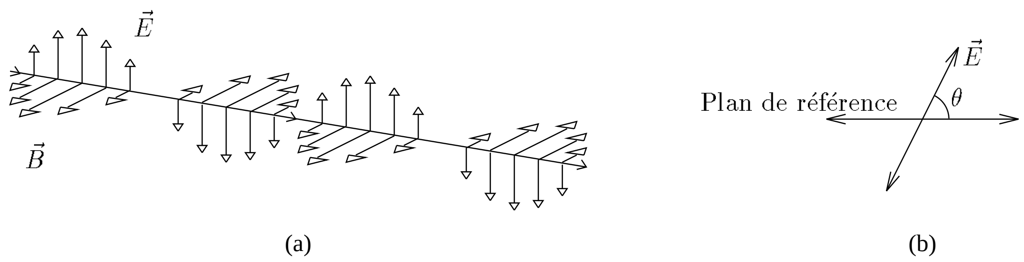

FIGURE 2.1. (a) Onde électromagnétique (b) polarisation

physique quantique, joue à peu près le même rôle qu'en algèbre linéaire. Un état quantique est en fait représenté par un *vecteur d'état*, soit un vecteur complexe de norme 1 (selon le produit scalaire standard sur les vecteurs complexes). Une chose à se rappeler est que le vecteur d'état est une représentation de l'état, selon une base bien spécifique. Si on change de base, le vecteur d'état changera.

Dans le cas de la polarisation, trois bases sont souvent utilisées:

- La base *rectilinéaire*, $\{|\rightarrow\rangle, |\uparrow\rangle\}$. C'est la base canonique pour exprimer la polarisation d'un photon, c'est-à-dire qu'elle correspond aux vecteurs $\{(1,0),(0,1)\}$.
- La base *diagonale*, $\{|\nearrow\rangle, |\nwarrow\rangle\}$. Exprimée en fonction de la première base, cela donne les vecteurs $\{(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}), (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})\}$.
- La base *circulaire*, $\{|\hookrightarrow\rangle, |\hookleftarrow\rangle\}$. Exprimée en fonction de la première base, cela donne les vecteurs $\{(\frac{1}{\sqrt{2}}, \frac{i}{\sqrt{2}}), (\frac{i}{\sqrt{2}}, \frac{1}{\sqrt{2}})\}$.

La notation $|\rangle$ n'est utilisée que pour différencier un vecteur d'état d'un vecteur complexe ordinaire. Par un abus de notation, le vecteur d'état décrivant la polarisation d'un photon α sera notée $|\alpha\rangle$. Une polarisation correspondant à un angle réel θ s'exprime, en fonction de la base canonique, sous la forme d'un vecteur $(\cos\theta, \sin\theta)$. Ces polarisations sont dites linéaires et ne forment qu'un sous-ensemble des polarisations exprimables.

Considérons maintenant la polarisation de 2 photons isolés α et β. Peut-on exprimer les 2 polarisations comme un seul vecteur d'état $|\alpha\beta\rangle$ (de dimension plus élevée)? La réponse est oui. Il suffit de choisir une base pour chacune des polarisations et de les combiner par un produit tensoriel, vecteur à vecteur. Par exemple, si chaque polarisation est exprimée dans la base rectilinéaire $\{|\rightarrow\rangle, |\uparrow\rangle\}$, alors les polarisations jointes seront exprimées dans la base $\{|\rightarrow\rightarrow\rangle, |\rightarrow\uparrow\rangle, |\uparrow\rightarrow\rangle, |\uparrow\uparrow\rangle\}$. Soit $|\alpha\rangle = (\alpha_0, \alpha_1)$ et $|\beta\rangle = (\beta_0, \beta_1)$, alors le vecteur de l'état joint sera

$$\begin{aligned}|\alpha\beta\rangle &= |\alpha\rangle \otimes |\beta\rangle \\ &= (\alpha_0\beta_0, \alpha_0\beta_1, \alpha_1\beta_0, \alpha_1\beta_1).\end{aligned}$$

Cette technique peut se généraliser à autant de polarisations jointes que l'on veut.

Maintenant, est-il possible de faire l'inverse, c'est-à-dire de séparer un vecteur exprimant la polarisation de plusieurs photons en plusieurs petits vecteurs d'état? La réponse est: pas toujours. En effet, il existe des vecteurs qui décrivent la polarisation de plusieurs photons et qui ne peuvent s'exprimer comme le produit tensoriel d'autres vecteurs. Le vecteur $(0, \frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0)$ en est un exemple. Ce vecteur d'état décrit une instance d'une paire E-P-R[1]. Un tel vecteur d'état est appelé *enchevêtré*, car les polarisations impliquées y sont trop liées pour être exprimées séparément. La dimension d'un vecteur d'état exprimant la polarisation de n photons est toujours égale à 2^n.

2.3. Transformation des états quantiques. Dans le monde réel, il y a plusieurs façons d'affecter la polarisation d'un photon:

- Passage à travers un filtre polarisant
- Réflexion partielle sur une substance réfringente
- Présence d'un champ magnétique ou électrique

Toutes les transformations sur les particules sont modélisées, au niveau quantique, par deux types d'opérations: les opérateurs unitaires et les mesures.

Un *opérateur unitaire* est analogue à une rotation *rigide* de la polarisation d'un ou plusieurs photons. Il s'exprime sous la forme d'une matrice carrée complexe telle que $U^\dagger U = I$. Par exemple, la matrice

$$U = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

est un opérateur unitaire qui fait tourner la polarisation d'un photon de $+45^o$ (ou $\frac{\pi}{4}$ radians). L'application d'un opérateur unitaire U à un état quelconque $|\psi\rangle$ est le produit matriciel $U|\psi\rangle$. Il est naturellement possible d'appliquer un opérateur unitaire sur un état à plusieurs photons. Il faut cependant que les dimensions soient consistantes entre la matrice et le vecteur d'état. Il est intéressant de noter que si U est un opérateur unitaire de dimensions $2^m \times 2^m$ et que V est un opérateur unitaire de dimensions $2^n \times 2^n$, alors le produit tensoriel des 2 matrices $U \otimes V$ est aussi un opérateur unitaire, de dimensions $2^{m+n} \times 2^{m+n}$. Ce nouvel opérateur consiste *grosso modo* à faire tourner les m premières polarisations avec U et les n dernières polarisations avec V. Autre fait intéressant: la matrice identité I_{2^k} est un opérateur unitaire qui laisse un vecteur d'état de k photons inchangé, une rotation de 0^o en quelque sorte.

[1]Un exemple de paire E-P-R (Einstein-Podolsky-Rosen) serait un état à 2 photons tel que toute mesure de polarisation faite sur une des particules fait en sorte que l'autre particule aura une polarisation inverse de la première.

Une *mesure quantique* [2] est définie par une base orthonormée de l'espace vectoriel dans lequel se trouve le vecteur d'état. L'acte de mesurer un état avec une base force son vecteur d'état à devenir un des éléments de la base. Le choix de l'élément est cependant aléatoire. Si $|\psi\rangle$ est le vecteur d'état et $\{\beta_0, \beta_1, \ldots, \beta_{2^n-1}\}$ est la base de mesure, alors la probabilité que $|\psi\rangle$ devienne l'élément $|\beta_i\rangle$ de la base est égale à

$$\|\langle \beta_i|\psi\rangle\|^2 = \sum_{j=0}^{2^n-1} \overline{\beta_{ij}}\psi_j$$

pour $i = 0 \ldots 2^n - 1$. La notation $\langle \beta_i|\psi\rangle$ est en fait le produit scalaire ordinaire entre vecteur complexes. Ainsi, plus un vecteur d'état est "proche" d'un des éléments de la base de mesure, plus il est probable que ce soit cet élément qui soit choisi. Le *résultat* de la mesure est l'indice i de l'élément choisi dans la base de mesure.

2.4. Le canal quantique. Le canal quantique, tel que présenté dans [2], [1], [5] et [10], peut être modélisé par ce diagramme:

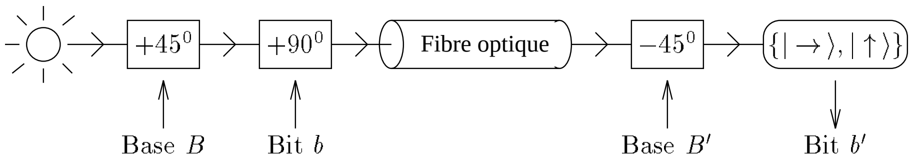

FIGURE 2.2. Modélisation du canal quantique

Le canal quantique s'utilise de la façon suivante:

1. Le participant à la source, S, possède un bit b qu'il désire envoyer à un participant récepteur, R, via le canal quantique.
2. S décide d'une base d'envoi $B \in \{0, 1\}$. Si $B = 0$, alors la base utilisée sera la base rectilinéaire. Sinon, ce sera la base diagonale.
3. R décide d'une base de réception $B' \in \{0, 1\}$, interprétée de façon similaire à B.
4. Un photon est émis d'une source lumineuse, polarisée horizontalement.
5. Si $B = 1$, une rotation de $+45^o$ est appliquée sur la polarisation du photon lorsqu'il passe à travers la première boîte. Sinon, le photon passe sans être modifié.
6. Si $b = 1$, une rotation de $+90^o$ est appliquée sur la polarisation du photon lorsqu'il passe à travers la deuxième boîte. Sinon, le photon passe sans être modifié.
7. Le photon est envoyé à R via un médium optique quelconque, comme par exemple une fibre optique.

[2] La physique quantique définit plusieurs genres de mesure, notamment les mesures complètes vs les mesures incomplètes et les mesures de von Neumann vs les mesures généralisées (POM ou IPP). Nous ne nous intéresserons qu'aux mesures de von Neumann complètes.

8. Si $B' = 1$, une rotation de -45^o est appliquée sur la polarisation du photon lorsqu'il passe à travers la troisième boîte. Sinon, le photon passe sans être modifié.
9. Pour finir, R mesure la polarisation du photon selon la base rectilinéaire et recueille le résultat dans le bit b'. Cette mesure est effectuée à l'aide d'un cristal de calcite [3] et de 2 photodétecteurs.

Si $B = B'$, alors le bit b' sera égal au bit b. Sinon, le bit b' prendra une valeur aléatoire. En effet, comme nous l'avons dit précédemment, plus la polarisation est proche d'un des éléments de la base, plus ce résultat est probable. Si $B = B'$, alors la polarisation sera obligatoirement un des vecteurs de la base de réception et la mesure sera parfaite. Sinon, la polarisation sera à mi-chemin entre les deux vecteurs de base B', ce qui rend équiprobables les deux résultats possibles de la mesure.

Pour tenir compte des erreurs de lecture et de transmission, un taux d'erreur cumulé ϵ est associé à une instance donnée du canal quantique. Ainsi, même si $B = B'$, il reste une probabilité ϵ que b' soit différent de b.

3. Simulation du modèle

La simulation du modèle quantique de la matière suppose une simulation des photons, des états quantiques, des opérateurs unitaires et des mesures. Commençons par les photons. Dans les manuels de physique, une particule est souvent désignée implicitement. La simulation d'une telle particule nécessite cependant un moyen de la désigner explicitement. Pour ce faire, nous utiliserons un nombre entier non-négatif pour désigner un photon, qui lui servira d'*étiquette*. Le nombre de particules existantes à l'exécution est limité par une constante configurable de **QSim**.

Un état quantique, dans le simulateur, sert à décrire la polarisation d'un ou de plusieurs photons. Ces photons doivent être bien spécifiés. Nous le ferons par une liste *ordonnée* des nombres entiers représentant les photons décrits par l'état. À cette liste, nous joignons un vecteur de nombres complexes représentant le vecteur d'état qui décrit la polarisation des photons de la liste. L'ordre de la liste est intimement lié à l'expression du vecteur d'état. Une permutation dans la liste des étiquettes induira une permutation dans les éléments du vecteur d'état. Dans la Figure 2.3, le premier bloc représente un état A, dont le vecteur d'état

[3]Le cristal de calcite est une substance biréfringente, c'est-à-dire qu'elle refracte la lumière polarisée horizontalement différemment de la lumière polarisée verticalement. Un faisceau de lumière polarisée horizontalement dirigé sur le cristal en sortirait en un endroit différent d'un faisceau de lumière polarisée verticalement dirigé sur le même emplacement.

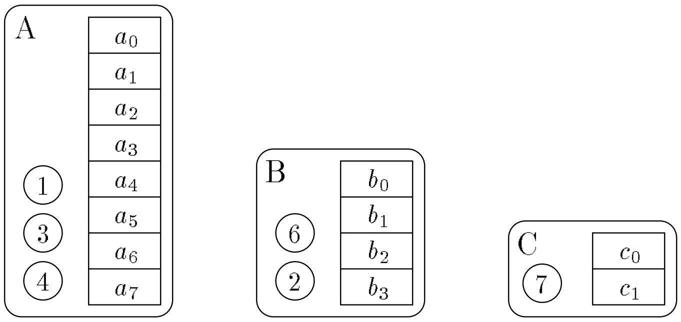

FIGURE 2.3. États quantiques simulés

$(a_0, \ldots, a_7)$ décrit la polarisation des photons étiquetés 1, 3 et 4. À côté , l'état B décrit la polarisation des photons 6 et 2, tandis que l'état C décrit la polarisation du photon 7.

La jonction de 2 états se fait:

- en concaténant les listes d'étiquettes des 2 états
- en prenant le produit tensoriel des 2 vecteurs d'états.

De plus, une permutation de la liste des étiquettes ne change rien à la nature physique de l'état. C'est un peu comme de regarder un objet sous différents angles. Ce sont toujours les mêmes polarisations qui y sont exprimées, mais pas dans le même ordre. C'est pourquoi le vecteur d'état doit être modifié afin de refléter cette permutation.

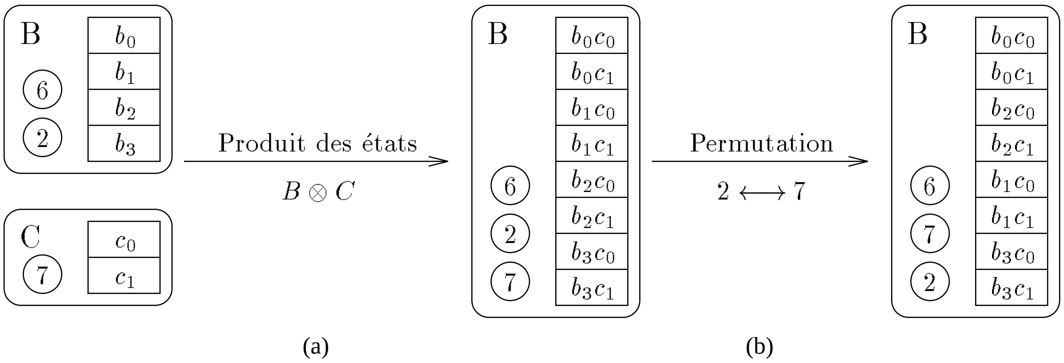

FIGURE 2.4. Exemples de (a) Fusion (b) Permutation

Soit U, un opérateur unitaire de dimension $2^n \times 2^n$. C'est un opérateur qui s'applique sur un groupe de n photons. Ce groupe est spécifié à l'opérateur sous la forme d'une liste d'étiquettes, un peu comme celles qui se trouvent dans un état. Le simulateur s'occupe de construire un état intermédiaire exprimant la polarisation de tous les photons spécifiés. Il utilisera l'algorithme de la Figure 2.5 pour la construction.

Soit $P[0\ldots n-1]$, la liste des étiquettes
Soit A, l'état auquel appartient l'étiquette $P[0]$

Permuter le photon $P[0]$ de sa position actuelle de la liste à la position 0.
pour $i \leftarrow 1$ **jusqu'à** $n-1$ **faire**
    Soit B, l'état auquel appartient $P[i]$
    si $A \neq B$ **alors** $A \leftarrow A \otimes B$ {jonction de A et B}
    {Le photon $P[i]$ fait maintenant partie de l'état A}
    Permuter le photon $P[i]$ de sa position actuelle de la liste à la position i.

retourner A

FIGURE 2.5. Algorithme pour la construction de l'état intermédiaire

Il peut arriver que la polarisation de certains de ces photons soit déjà enchevêtrée avec celle d'autres photons. Dans ce cas, il faudra inclure ces nouveaux venus dans l'état intermédiaire. Soit 2^m, la dimension de cet état intermédiaire. La matrice de l'opérateur unitaire devra être transformée en une matrice de dimensions $2^m \times 2^m$, qui ne s'appliquera que sur les n premiers photons de l'état intermédiaire. Un tel opérateur est obtenu par le produit tensoriel de U avec la matrice identité $I_{2^{m-n}}$. Ce nouvel opérateur est ensuite appliqué de façon normale sur l'état intermédiaire.

L'application d'une mesure se fait de façon similaire. Soit $B = \{\beta_0, \beta_1, \ldots, \beta_{2^n-1}\}$, la base de mesure. Ceci spécifie une mesure s'appliquant sur un groupe de n photons. Comme pour un opérateur unitaire, ce groupe est spécifié sous la forme d'une liste d'étiquettes. Le simulateur construit l'état intermédiaire de la même façon. Soit 2^m, la dimension de l'état intermédiaire. La base de mesure devra être transformée en une base pour cet espace de dimension plus élevée. Dans le cas de l'opérateur, ceci se faisait par un produit tensoriel avec une matrice identité. Ici, cela va se faire par un produit tensoriel avec la base canonique pour les états de $m-n$ photons. La mesure force l'état intermédiaire à devenir un des états de cette nouvelle base. Soit β'_i, cet état. Pour ramener i dans l'intervalle $0..2^n-1$, il nous faut retrouver de quel photon de la base originale β'_i fut dérivé. À cause de la façon dont nous construisons cette base, chaque vecteur de la base originale induit 2^{m-n} vecteurs consécutifs dans la nouvelle base, pour un total de 2^m vecteurs. Le résultat de la mesure sera $\lfloor \frac{i}{2^{m-n}} \rfloor$.

La simulation du canal quantique peut se faire de 2 façons: une version longue et une version rapide. La version longue n'est, en fait, pas implantée directement. Elle consiste à la simulation pas-à-pas du canal quantique décrit à la section précédente, en utilisant les opérateurs unitaires et la mesure finale. La version rapide permet de faire un grand nombre de transferts quantiques en une seule opération de simulation. Dans la version rapide, S fournit un vecteur de n bits représentant les bases utilisées pour chaque transfert unitaire (0 = base rectilinéaire et 1 = base diagonale) et un vecteur de n bits à envoyer à R. De son côté, R fournit un vecteur de n bits pour ses bases. À la fin, R reçoit un vecteur de n bits, contenant les bits qu'il a reçus par le canal quantique.

Notre implantation a cependant ses limites. Un état exprimant la polarisation de k photons occupent environ 2^{k+4} octets de mémoire. À cause des capacités des ordinateurs d'aujourd'hui, nous ne pouvons définir des états ayant plus de 28 particules enchevêtrées simultanément, ce qui est bien peu. En fait, déjà avec un état à 16 particules, la simulation devient très lente. Ces limites sont cependant suffisantes pour une implantation *honnête* des principaux protocoles utilisant la simulation quantique. Par honnête, nous entendons que les partenaires s'en tiennent au protocole établi et ne cherchent pas à obtenir de l'information supplémentaire.

4. Un algorithme de factorisation d'états

Considérons un état à 6 photons. La mémoire qu'il occupe est d'environ $2^{10} = 1024$ octets. Par contre, 2 états de 3 photons occupent environ $2 \times 2^7 = 256$ octets. Ceci est 4 fois moins de mémoire pour le même nombre de particules. Ce gain fut notre principale motivation pour

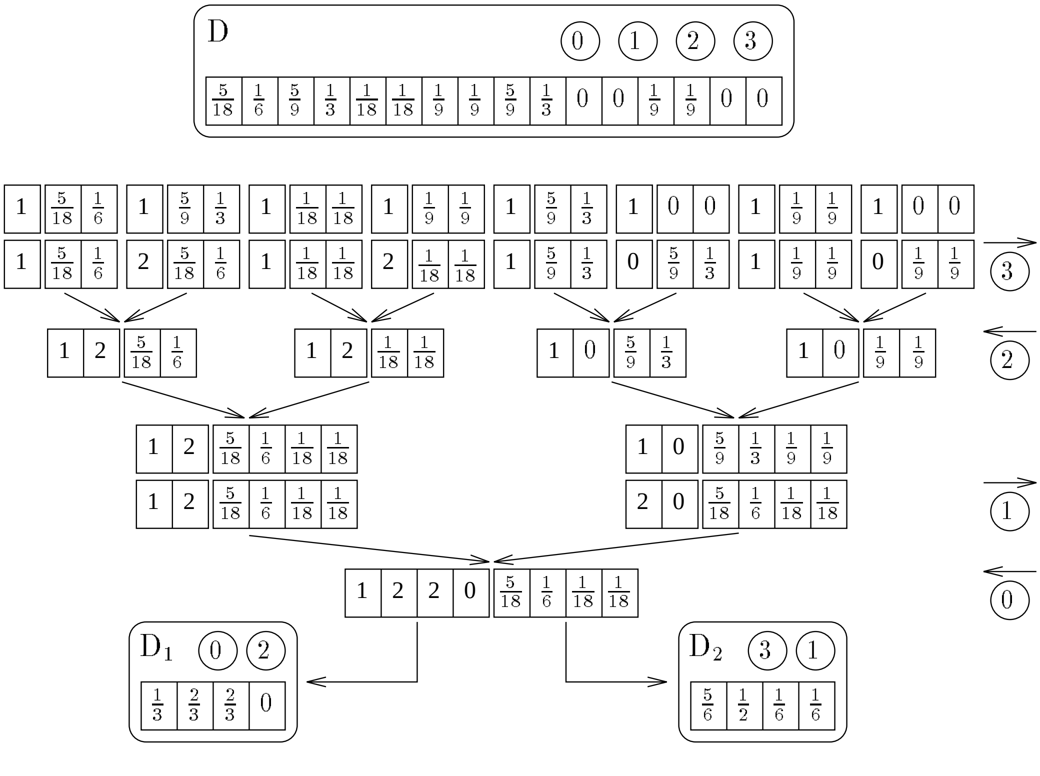

Figure 2.6. Factorisation d'un état

trouver une méthode efficace de séparer un gros état en plusieurs petits états. C'est ce que nous entendons par factorisation d'états. La Figure 2.6 nous en donne un exemple.

Soit $D = (P[0.\ldots.n-1], v[0.\ldots.2^n-1])$, l'état que nous voulons factoriser.
P est la liste des étiquettes de photons de l'état, tandis que v est le vecteur d'état.
Nous voulons retourner $D_1 = (P_1, v_1)$ et $D_2 = (P_2, v_2)$ tels que $P_1 \otimes P_2$ soit une permutation de P.
Soit T, une liste de couples de listes. T' sera une version de travail de T.

$P_1 \leftarrow \{\}$
$P_2 \leftarrow \{P[n-1]\}$
pour $i \leftarrow 0$ **jusqu'à** $2^{n-1}-1$ **faire**
  $T[i][0] \leftarrow \{1\}$
  $T[i][1] \leftarrow \{v[2i], v[2i+1]\}$

pour $i \leftarrow 2$ **jusqu'à** n **faire**
  pour $j \leftarrow 0$ **jusqu'à** 2^{n-i} **faire**
    si il existe $a, b \in C$, telle que $ab \neq 0$ et $aT[2j][1] = bT[2j+1][1]$ **alors**
      $\{T[2j]$ et $T[2j+1]$ sont proportionnels *à droite*$\}$
      $T'[j][0] \leftarrow bT[2j][0]$ concaténé avec $aT[2j+1][0]$
      si $T[2j+1][1]$ est le vecteur nul **alors**
        $T'[j][1] \leftarrow T[2j][1]$
      sinon
        $T'[j][1] \leftarrow T[2j+1][1]$
      $P_1 \leftarrow \{P[n-i]\} \cup P_1$
    sinon si il existe $a, b \in C$, telle que $ab \neq 0$ et $aT[2j][0] = bT[2j+1][0]$ **alors**
      $\{T[2j]$ et $T[2j+1]$ sont proportionnels *à gauche*$\}$
      $T'[j][1] \leftarrow bT[2j][1]$ concaténé avec $aT[2j+1][1]$
      si $T[2j+1][0]$ est le vecteur nul **alors**
        $T'[j][0] \leftarrow T[2j][0]$
      sinon
        $T'[j][0] \leftarrow T[2j+1][0]$
      $P_2 \leftarrow P_2 \cup \{P[n-i]\}$
    sinon
      Rejeter, car l'état n'est pas factorisable
  $T \leftarrow T'$

$v_1 \leftarrow T[0]$ normalisé
$v_2 \leftarrow T[1]$ normalisé

FIGURE 2.7. Algorithme pour la factorisation d'états

Les gros états dont nous parlons sont en fait les états intermédiaires construits lors de l'application d'un opérateur unitaire ou d'une mesure. Étant donné qu'ils regroupent plusieurs états, ils occupent de plus en plus de mémoire. C'est pourquoi il serait intéressant de séparer ces états intermédiaires en plusieurs petits états. Nous allons maintenant présenter un algorithme efficace, résultat de nos recherches, pour vérifier si un état est un produit, et si c'est le cas de trouver ses états *facteurs*. La Figure 2.7 décrit cet algorithme.

Notez que les tests d'égalité entre nombres complexes sont implantés en comparant la norme de leur différence avec un petit nombre ϵ fixé d'avance.

L'algorithme commence par convertir le vecteur d'état en paires de vecteurs. À chaque étape, les paires de vecteurs sont jointes, deux à deux, sous certaines conditions. À la fin, il ne reste qu'une paire de vecteurs, représentant les deux états facteurs désirés.

L'algorithme peut échouer de 2 façons: en rejettant, ou si $D_2 = D$. Dans les 2 cas, cela indique que l'état D ne peut être factorisé. Lorsque l'algorithme réussit à trouver 2 états facteurs D_1 et D_2, l'état D_2 est toujours irréductible. Cet algorithme prend un temps dans $O(n2^n)$. Ceci peut ne pas sembler très efficace, mais il ne faut pas oublier que la longueur de l'entrée D est de $n + 2^n$.

5. Références

L'introduction sur la polarisation des photons fut tirée de [23].

Le concept de paire de particules E-P-R est apparu dans [15].

CHAPITRE 3

STRUCTURES DE DONNÉES

1. Introduction

QSim se présente comme étant un ensemble de structures de données et d'opérations sur ces structures, écrites en C++, facilitant la simulation de protocoles cryptographiques. Ces structures de données sont très variées et devraient couvrir la plupart des besoins de simulation. Cependant, aucune de ces structures n'est véritablement une invention. Toutes sont bien connues. Le seul aspect créatif concernent la façon dont elles ont été implantées. Pour chacune de ces structures de données, nous utiliserons le même format descriptif, soit une courte description utilitaire du type, une liste des opérations usuelles associées à ce type et certaines particularités d'implantation. Les opérations sur les types sont énumérées simplement, sans décrire les paramètres ni le type de retour. Elles ne sont présentées que pour donner une meilleure compréhension de ces types. Pour certaines d'entre elles, un commentaire entre parenthèses expliquera le rôle qu'ont ces opérations. Celles qui n'en ont pas ont leur sens usuel.

2. Types élémentaires

Les types élémentaires sont `Char`, `Bit`, `Byte`, `Integer`, `Longint`, `Boolean`, `Real` et `Void`. Ce sont les types les plus simples. Les autres types sont construits à partir de ces types élémentaires.

2.1. Type `Char`. Ce type permet de décrire les caractères ASCII. Ce type peut aussi être manipulé comme un type entier. Contrairement au type `char` du langage PASCAL, nous n'avons pas besoin de fonction de conversion entre caractères et entiers. Les opérations usuelles sur ce type sont:

- `++` (Successeur), `--` (Prédeceseur)
- `+`, `-` (Déplacement dans la table ASCII)
- `==` (Égalité), `!=` (Inégalité), `<`, `<=`, `>`, `>=`

2.2. Types entiers. Les types `Bit`, `Byte`, `Integer` et `Longint` sont les types entiers. Le type `Bit`, comme son nom l'indique, ne peut prendre que les valeurs 0 et 1. Le type `Byte` est un entier de 8 bits, sans signe, permettant de représenter des valeurs comprises entre 0 et 255. Le type `Integer` est un entier de 16 bits, avec signe, permettant de représenter des valeurs comprises entre -32768 et 32767. Le type `Longint` est un entier de 32 bits, avec signe, permettant de représenter des valeurs comprises entre -2147483648 et 2147483647. Les opérations usuelles sur les entiers sont:

- `++` (Successeur), `--` (Prédecesseur)
- `-` (Unaire)
- `+`, `-`, `*`, `/`, `%` (Modulo)
- `<<`, `>>` (Décalage des bits du nombre)
- `&` (Et binaire), `|` (Ou binaire), $\wedge$ (Ou-exclusif binaire), $\sim$ (Complément)
- `==`, `!=`, `<`, `<=`, `>`, `>=`
- `randomBit`, `randomByte`, `randomInteger`, `randomLongint` (Génération *aléatoire*)

Le cas du type `Bit` est un peu spécial: `+` et `-` ont tous deux le même sens que $\wedge$, et `*` a le même sens que `&`. Les fonctions de génération aléatoire utilisent un générateur pseudo-aléatoire basé sur la congruence linéaire, implanté par la fonction `lrand48`. Le nombre 48 indique le nombre de bits impliqués dans le calcul du prochain nombre pseudo-aléatoire. Seuls les 32 premiers bits de ce nombre sont retournés. Un générateur basé sur la physique quantique serait beaucoup plus aléatoire. La mesure d'un photon polarisé comme $|\nearrow\rangle$, mesuré dans la base rectilinéaire, retournerait un bit parfaitement aléatoire.

2.3. Type `Boolean`. Ce type permet de représenter des valeurs booléennes. Les noms des valeurs de vérité sont `TRUE` et `FALSE`. Il est possible d'initialiser un `Boolean` à partir d'une valeur entière. La convention est celle du C (et du C++) standard: 0 équivaut à `FALSE` et le reste équivaut à `TRUE`. Les opérations usuelles sur ce type sont:

- `&&` (Et logique), `||` (Ou logique), `!` (Négation)
- `==`, `!=`
- `randomBoolean` (Génération aléatoire)

La même remarque que pour les types entiers s'applique sur le générateur aléatoire pour booléens.

2.4. Type `Real`. Ce type permet d'exprimer des nombres réels. Les valeurs réelles sont représentées sur 64 bits. La plus petite valeur représentable 2.2250738585072014E$-$308, tandis que la plus grande valeur est 1.7976931348623157E+308. Les opérations usuelles sur les réels sont:

- `-` (Unaire)
- `+`, `-`, `/`, `*`
- `==`, `!=`, `<`, `<=`, `>`, `>=`
- `sqrt` (Racine carrée), `log` (Logarithme naturel), `exp` (Exponentielle)
- `randomReal` (Génération aléatoire)

Le générateur aléatoire pour les réels est implanté par la fonction `drand48`, qui retourne un nombre en point flottant uniformément choisi dans l'intervalle $[0, 1)$.

2.5. Type Void. `Void` n'est pas un véritable type. Il sert à décrire le type de retour d'une fonction ne retournant aucun résultat (comme une procédure en PASCAL).

3. Algèbre linéaire

Les types décrits dans cette section implantent les structures mathématiques retrouvées en algèbre linéaire, telles que les polynômes, les vecteurs et les matrices.

3.1. Type `Array`. Ce type sert à exprimer une collection ordonnée de taille connue d'objets de même type. C'est un tableau à une dimension, avec possibilité d'expansion, dont les indices commencent à 0. Il ne possède aucune propriété algébrique particulière, mais sert de base aux types `Poly` et `Vector`. Les opérations usuelles sur les `Array` sont:

- `length` (Taille du tableau)
- `find` (Recherche d'un élément dans le tableau)
- `reshape` (Redimensionnement du tableau)
- `fill` (Remplissage du tableau)
- `sub` (Accès à une portion du tableau)
- `[]` (Accès à un élément du tableau)
- `&&` (Concaténation de 2 tableaux)
- `==`, `!=`

Toutes les opérations qui tenteraient d'accèder en dehors des bornes du tableau provoqueraient une erreur. C'est le cas des opérations `[]` et `sub`.

3.2. Type `Poly`. Ce type est un `Array` un peu spécial, dont l'accès du côté des indices positifs est *infini*[1]. Il sert à représenter des polynômes. Il possède toutes les opérations usuelles du type `Array`, en plus de celles-ci:

[1]La longueur d'un polynôme n'est limitée que par la mémoire de l'ordinateur.

- `deg` (Degré du polynôme)
- `()` (Évaluation du polynôme en un point)
- `-` (Unaire), `*` (Par un scalaire)
- `+`, `-`, `*`, `/`, `%` (Modulo)

Comme cette structure de données est infinie du côté des indices positifs, l'opération `[]` accepte n'importe quel indice positif. Un indice négatif est cependant hors des bornes et provoquera une erreur. L'opération `deg` retourne -1 dans le cas d'un polynôme nul. La fonction `length` fonctionne toujours, mais retournera une longueur qui ne correspondera pas nécessairement à $1 +$ `deg`.

3.3. Type `Vector`. Ce type est un autre `Array`, spécialisé dans les espaces vectoriels. Il possède toutes les opérations usuelles du type `Array` en plus de celles-ci:

- `hamming` (Distance de Hamming entre 2 vecteurs)
- `weight` (Poids de Hamming d'un vecteur)
- `is_zero` (Test de nullité d'un vecteur)
- `sum` (Somme des éléments d'un vecteur)
- `-` (Unaire), `*` (Par un scalaire)
- `+`, `-`, `*`, `/` (Élément à élément)
- `>>`, `<<` (Décalage des éléments)
- $\wedge$ (Produit scalaire entre 2 vecteurs)

La distance de Hamming entre 2 vecteurs est le nombre de positions où leurs entrées sont différentes. Le poids de Hamming d'un vecteur est le nombre d'entrées non nulles.

3.4. Type `Matrix`. Ce type est le pendant à 2 dimensions du type `Vector`. Certaines de ses opérations impliquent un objet de type `Vector`. Voici les opérations usuelles du type `Matrix`:

- `rows` (Nombre de rangées), `cols` (Nombre de colonnes)
- `reshape` (Redimensionnement)
- `fill` (Remplissage)
- `inv` (Inverse d'une matrice carrée)
- `row` (Copie d'une rangée), `col` (Copie d'une colonne)
- `[]` (Indiçage simple)
- `[][]` (Indiçage double)
- $\wedge$ (Produit matriciel)
- `==`, `!=`

L'inversion d'une matrice non-carrée provoquera une erreur. L'indiçage simple dans une matrice retourne la rangée correspondante dans la matrice, selon les conventions mathématiques habituelles. L'indiçage double permet d'accéder à un élément précis de la matrice. Comme pour le type `Array`, si les indices sortent des bornes de la matrice, cela provoquera une erreur. C'est aussi le cas pour les opérations `row` et `col`. Le produit matriciel peut se faire entre un vecteur ou une matrice, ou entre 2 matrices. Il faut cependant que leurs dimensions soient consistantes, sinon il y aura erreur.

3.5. Vecteurs et matrices de bits. Les types `Array`, `Poly`, `Vector` et `Matrix` sont polymorphiques, pouvant prendre comme type de leurs éléments n'importe quel autre type. Lorsque ce type est le type `Bit`, les types `Vector` et `Matrix` ont quelques opérations supplémentaires:

- $\sim$ (Complément d'un vecteur de bits)
- `parity` (Parité d'un vecteur de bits)
- `filter` (Extraction et concaténation de certains bits d'un vecteur en fonction d'un masque)
- `integer_value` (Conversion d'un vecteur de bits à l'entier d'expansion binaire correspondante)
- `randomize` (*Randomisation* des éléments d'un vecteur ou d'une matrice de bits)

3.6. Type `Galois`. Ce type est un vecteur de bits spécialisé permettant de représenter les éléments d'un corps fini de caractéristique 2. Ces corps sont mieux connus sous la désignation GF(2^n). Les opérations usuelles sur ces éléments (en plus des opérations sur les vecteurs de bits) sont:

- `inv` (Inverse multiplicatif)
- `next_power` (Multiplication par 2 rapide), `prev_power` (Division par 2 rapide)
- `+`, `-`, `*`, `/`

La somme, tout comme la différence, de 2 éléments dans le corps est le ou-exclusif terme-à-terme des bits de ces éléments. Le produit de 2 éléments dans le corps est la multiplication polynomiale des 2 éléments vus comme des polynômes de bits, modulo un polynôme réducteur primitif de degré $n+1$ pour ramener le résultat à n bits. Ce polynôme est extrait d'une table pré-calculée et indexée par le degré du polynôme. Pour l'instant, cette table est d'une longueur limitée à 128. Une telle limite satisfait nos besoins pour l'instant. De plus, les polynômes réducteurs primitifs commencent à être difficile à calculer pour de telles longueurs. Nous avons utilisé le logiciel mathématique Maple pour faire nos calculs et trouver ces polynômes. Dans le cas d'une multiplication ou d'une division par 2, il est plus efficace d'utiliser `next_power` et `prev_power` que `*` et `/`.

3.7. Type `GF`. Le type `GF` a été ajouté pour faciliter la génération d'éléments d'un corps fini tels qu'implantés par le type `Galois`. Il représente en quelque sorte le corps au complet. Voici les opérations usuelles sur ce type:

- `dim` (Dimension du corps)
- `zero` (Valeur de l'élément neutre additif du corps)
- `unity` (Valeur de l'élément neutre multiplicatif du corps)
- `root` (Valeur d'une racine principale de l'unité du corps)
- `random` (Génération aléatoire d'un élément dans le corps)
- `element` (Valeur correspondant, dans le corps, à l'expansion binaire d'un nombre entier)

L'opération `dim` retourne n, c'est-à-dire le logarithme en base 2 de la cardinalité de l'ensemble. Ceci correspond aussi au nombre de bits des éléments du corps. L'opération `root` retourne toujours 2 (ou plutôt son équivalent en vecteur de bits $010\ldots0$), car dans le cas particulier des corps $\mathrm{GF}(2^n)$, c'est toujours une bonne racine principale de l'unité. Tenter de calculer l'inverse de la valeur `zero` ou diviser par cette même valeur provoquera une erreur.

4. Codes correcteurs linéaires

Imaginez la situation suivante: Vous désirez transmettre un message de k caractères pris dans un alphabet A via un canal bruyant. Ce canal n'est pas fiable car, avec une probabilité $p < \frac{1}{2}$, le caractère émis sera modifié par le canal. Pour vous protéger contre ces erreurs, vous devrez ajouter de la redondance à votre message, pour compenser l'effet destructeur du bruit. La façon la plus générale de définir cette redondance est d'associer à chaque message possible un autre message de $n > k$ caractères généralement pris dans le même alphabet. À chaque message original correspond un unique message plus long. Ceci définit de façon naturelle une fonction d'*encodage* $C : A^k \to A^n$, inversible. L'image de cette fonction est appelée un *code correcteur*. Voici quelques définitions relatives aux codes correcteurs:

- Nous appelons *mots d'information*, les messages originaux de k caractères.
- Nous appelons *mots de code*, les images de ces messages par la fonction d'encodage C.
- k, la longueur des messages originaux, est appelée la *dimension* du code.
- n, la longueur des mots de code, est appelée la *longueur* du code.
- Pour chaque paire de mots de même longueur, nous pouvons calculer le nombre de positions où ils sont différents. Nous appelons ce nombre la *distance de Hamming* entre les mots.

- Considérons la distance de Hamming entre chaque paire de mots du code. La valeur minimale de cette distance est appelée *distance minimale* du code, notée d. Plus cette valeur est grande, plus il est théoriquement possible de corriger d'erreurs.
- Un code correcteur est souvent désigné par ses paramètres $[n, k, d]$.

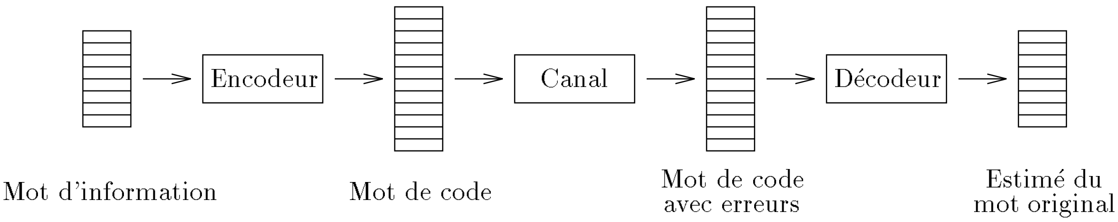

FIGURE 3.1. Transmission sur un canal bruyant

Maintenant, vous envoyez ce message redondant sur le canal bruyant. Il y a de très fortes chances que le message reçu soit différent du message émis. Ceci n'est pas grave en autant que la redondance suffise à corriger ces différences. La fonction d'encodage doit avoir été choisie selon 3 critères:

- Il faut que la distance minimale du code soit suffisament élévée pour que le décodage soit possible. Étant donné que le nombre d'erreurs prévues est de np, une valeur de d supérieure à $2np + 1$ nous permettrait, avec une bonne probabilité, de corriger la plupart des mots passant par le canal.
- Il faut que le retour au mot d'information à partir du message reçu se calcule de façon efficace.
- Le paramètre n ne devrait pas être trop grand par rapport au paramètre k.

Ceci décrit de façon générale les codes correcteurs. Un grand nombre des codes utilisés aujourd'hui appartiennent à un sous-ensemble de ces codes: ce sont les codes correcteurs linéaires. Un code correcteur linéaire est tout simplement un code correcteur ayant les propriétés d'un espace vectoriel. Ceci implique que le vecteur $\vec{0}$ est toujours un mot de code et que la somme de 2 mots de code est aussi un mot de code. Un code linéaire est souvent défini comme le noyau d'une matrice H de dimensions $n \times (n-k)$, appelée *matrice de parité* du code (pour des raisons historiques). Ainsi, pour tout mot du code c, le produit matriciel Hc^T donne le vecteur $\vec{0}$, et pour tout autre mot de dimension n, le même produit donne un vecteur non nul. Ce produit est appelé le *syndrome* associé au mot. Un autre aspect intéressant des codes linéaires est la possibilité d'exprimer la fonction d'encodage sous la forme d'une *matrice génératrice* G de dimensions $n \times k$. L'espace vectoriel engendré par les colonnes de G est le code correcteur.

Tous les codes correcteurs implantés dans **QSim** sont linéaires. Et chacun de ces codes offre la même brochette d'opérations:

- `n` (Longueur des mots du code)
- `k` (Dimension du code)
- `d` (Distance minimale du code)
- `encode` (Mot de code associé à un mot d'information donné)
- `random_codeword` (Génération d'un mot de code aléatoire)
- `random_word` (Génération d'un mot aléatoire d'un syndrome donné)
- `syndrome` (Syndrome associé à un mot donné)
- `is_codeword` (Test d'appartenance d'un mot donné au code)
- `correct` (Mot de code le plus proche d'un mot donné)
- `decode` (Mot d'information générant le mot de code le plus proche d'un mot donné)

Dans le cas des opérations `correct` et `decode`, une valeur booléenne est aussi retournée indiquant si l'opération a réussi.

4.1. Codes binaires. Dans un code linéaire binaire $[n, k, d]$, les mots d'information sont des vecteurs de k bits, les mots de code sont des vecteurs de n bits et chaque paire de mots de code compte au moins d positions où les bits sont différents. Les codes binaires implantés dans **QSim** utilisent un encodage dit systématique, c'est-à-dire que l'encodage d'un mot d'information de k bits ajoute à la fin de ce mot $n-k$ bits de redondance. L'extraction du mot d'information à partir d'un mot de code quelconque est ainsi très facile, car on ne garde que les k premiers bits.

4.1.1. *Type* `RandomCode`. Ce type implante les codes binaires aléatoires, c'est-à-dire dont la matrice de parité est choisie aléatoirement parmi toutes les matrices binaires de dimensions $n \times (n-k)$. Cette classe de codes binaires est utile seulement dans les cas où on doit encoder des mots sans chercher à les décoder. En effet, les opérations de recherche de distance minimale et de décodage peuvent prendre un temps excessif avec des codes de grande taille, car la meilleur technique connue pour résoudre ces problèmes sur ces codes est la recherche exhaustive.

Cependant, ce type de code binaire n'est pas sans avantage. Les codes binaires aléatoires sont asymptotiquement bons en moyenne, c'est-à-dire que lorsque n devient grand, à un taux de transmission constant k/n correspond un rapport d/n constant (en moyenne) suivant la loi de Varshamov-Gilbert:

$$H_2\left(\frac{d}{n}\right) \sim 1 - \frac{k}{n}$$

où H_2 est la fonction d'entropie en base 2.

4.1.2. *Type* `HammingCode`. Ce type implante les codes de Hamming binaires. Les codes de Hamming possèdent trois propriétés très intéressantes:

- La distance minimale d'un code de Hamming est toujours 3.
- La j^e colonne de la matrice de parité d'un code de Hamming $[n, k, 3]$ est l'expansion binaire de l'entier j sur n bits.
- Si un mot contient une seule erreur, le syndrome de ce mot est en fait l'expansion binaire de la position erronnée. La correction d'une seule erreur est donc très efficace.

Voici un exemple des 2 derniers points. Considérons le code de Hamming [7,3,3] binaire. Sa matrice de parité est:

$$H = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Soit $c = [1\,0\,1\,1\,0\,1\,0]$, un mot. Le calcul du syndrome nous assure que c'est un mot du code. Soit $c' = [1\,0\,0\,1\,0\,1\,0]$, égal à c, sauf en position 3. Multiplions H par ${c'}^T$:

$$H{c'}^T = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Le résultat est bel et bien l'expansion binaire de 3 sur 3 bits. Les valeurs des paramètres pour lesquels les codes de Hamming sont implantés sont les suivantes. Si $2 \leq m \leq 5$, alors

- $2^{m-1} < n \leq 2^m - 1$
- $k = n - m$

Les codes de Hamming sont donc implantés pour des valeurs de n entre 3 et 31.

4.2. Codes de Reed-Solomon. Le type `RSCode` est une implantation des codes de Reed-Solomon. Dans un tel code, les mots d'informations sont des polynômes sur un corps F de degré inférieur à k, et les mots du code sont des polynômes sur le même corps F de degré inférieur à n. Dans cette implantation, les corps utilisés sont les GF(2^n) implantés par les types `Galois` et `GF`. Si nous fixons la cardinalité du corps à $N = 2^n$, nous obtenons un code de Reed-Solomon pour chaque valeur de K comprise entre 1 et N, dont les paramètres sont $[N-1, K, N-K]$.

Contrairement aux codes binaires décrits précédemment, l'encodage n'est pas fait de façon systématique. Soit α, une racine primitive de l'unité pour le corps utilisé[2]. Soit un mot

[2]Dans le cas des corps GF(2^n), l'expansion binaire de l'entier 2 sur n bits est toujours une bonne racine primitive de l'unité, car les polynômes réducteurs utilisés sont primitifs.

d'information, $u = (u_0, u_1, \ldots, u_{K-1})$, et soit

$$u(z) = \sum_{i=0}^{K-1} u_i z^i.$$

Alors

$$c = (u(1), u(\alpha), u(\alpha^2), \ldots, u(\alpha^{N-1})).$$

Le calcul du syndrome s d'un mot c se fait d'une façon similaire. Si $c = (c_0, c_1, \ldots, c_{N-1})$, et si

$$c(x) = \sum_{i=0}^{N-1} c_i x^i,$$

alors

$$s = (c(\alpha), c(\alpha^2), c(\alpha^3), \ldots, c(\alpha^{N-K})).$$

La correction des erreurs se fait par l'algorithme de Berlekamp pour la correction d'erreurs dans les codes alternants[3]. Sur la Figure 3.2, vous trouverez une description en pseudo-code de cet algorithme. La preuve du fonctionnement de cet algorithme sort malheureusement du cadre

Soit c, le mot à corriger et s, le syndrome de c.
Soit q, r_0, r_1, u_0, u_1, r' et u', des polynômes.
Soit marque$[0..N-1]$, un tableau de booléens.

{Algorithme d'Euclide pour le PGCD de 2 polynômes}
$r_0 \leftarrow s$; $r_1 \leftarrow x^{N-K}$; $u_0 \leftarrow 1$; $u_1 \leftarrow 0$
$i \leftarrow 0$
tantque $\deg r_i \geq \frac{1}{2}(N-K)$ **faire**
    $q \leftarrow r_{1-i}/r_i$
    $r_{1-i} \leftarrow r_{1-i}$ **mod** r_i
    $u_{1-i} \leftarrow u_{1-i} + qu_i$
    $i \leftarrow 1 - i$

$r' \leftarrow (u_i(0))^{-1} r_i$; $u' \leftarrow (u_i(0))^{-1} u_i$
pour $j \leftarrow 0$ **jusqu'à** $N-1$ **faire** marque$[j] \leftarrow (u'(\alpha^j) = 0)$

pour $j \leftarrow 0$ **jusqu'à** $N-1$ **faire**
    si marque$[j]$ **alors**
        $g \leftarrow 1$
        pour $l \leftarrow 0$ **jusqu'à** $N-1$ **faire**
            si marque$[l]$ **et** $(j \neq l)$ **alors** $g \leftarrow g \times (1 + \alpha^{-l}\alpha^j)$
    $c_{N-j} \leftarrow c_{N-j} + \frac{\alpha^{-j} r'(\alpha^j)}{g}$

retourner c

FIGURE 3.2. Algorithme de Berlekamp

[3]Un code *alternant* est une généralisation des codes de BCH, eux-mêmes une généralisation des codes de Reed-Solomon.

de ce mémoire. Une fois le mot de code le plus proche c trouvé, il est possible de retrouver le mot d'information correspondant u. Si $c = (c_0, c_1, \ldots, c_{N-1})$, et si

$$c(x) = \sum_{i=0}^{N-1} c_i x^i,$$

alors

$$u = (c(1), c(\alpha^{-1}), c(\alpha^{-2}), \ldots, c(\alpha^{-K+1})).$$

4.3. Codes concaténés. Il est possible de combiner un code de Reed-Solomon avec un code binaire si la dimension du code binaire est égale à la dimension du corps fini utilisé dans le code de Reed-Solomon. Nous obtenons ainsi un exemple de code concaténé. Voici quelques définitions utiles sur les codes concaténés:

- Nous appellerons *code externe* le code de Reed-Solomon, car ce code n'est impliqué qu'aux extrémités du processus de transmission. En effet, l'encodage par ce code se fait en tout premier lieu, tandis que le décodage par ce code est la toute dernière étape.
- Nous appellerons *code interne* le code binaire, par opposition au code externe.
- n_E dénotera la longueur du code externe.
- k_E dénotera la dimension du code externe.
- n_I dénotera la longueur du code interne.
- k_I dénotera la dimension du code interne.
- Les mots d'information sont des vecteurs de $k_E k_I$ bits, ce qui fait que $k = k_E k_I$.
- Les mots de code sont des vecteurs de $n_E n_I$ bits, ce qui fait que $n = n_E n_I$.
- Une conséquence de cette construction est que $d \geq d_E d_I$. Cependant, la méthode que nous utiliserons pour corriger un mot ne permettra que de corriger $\frac{d_E d_I}{4}$ erreurs.

L'encodage d'un mot d'information n'est pas systématique et se fait en 3 étapes, comme nous pouvons le voir sur la partie gauche de la Figure 3.3 (de haut en bas):

1. Premièrement, nous découpons le mot d'information u en k_E morceaux de k_I bits, en transformant chaque petit morceau en un élément du corps fini.
2. Deuxièmement, nous pouvons transformer le mot d'information en un polynôme sur le corps fini, de degré inférieur à k_E, et l'encoder selon le code externe.
3. Ceci nous donne n_E éléments du corps fini. Chacun de ces éléments est retransformé en un vecteur de k_I bits et encodé selon le code interne.

Nous avons ainsi n_E petits vecteurs de n_I bits, formant au total un vecteur de $n_E n_I$ bits, c.

Le calcul du syndrome est légérement plus tordu (partie droite la Figure 3.3) et se fait aussi en 3 étapes:

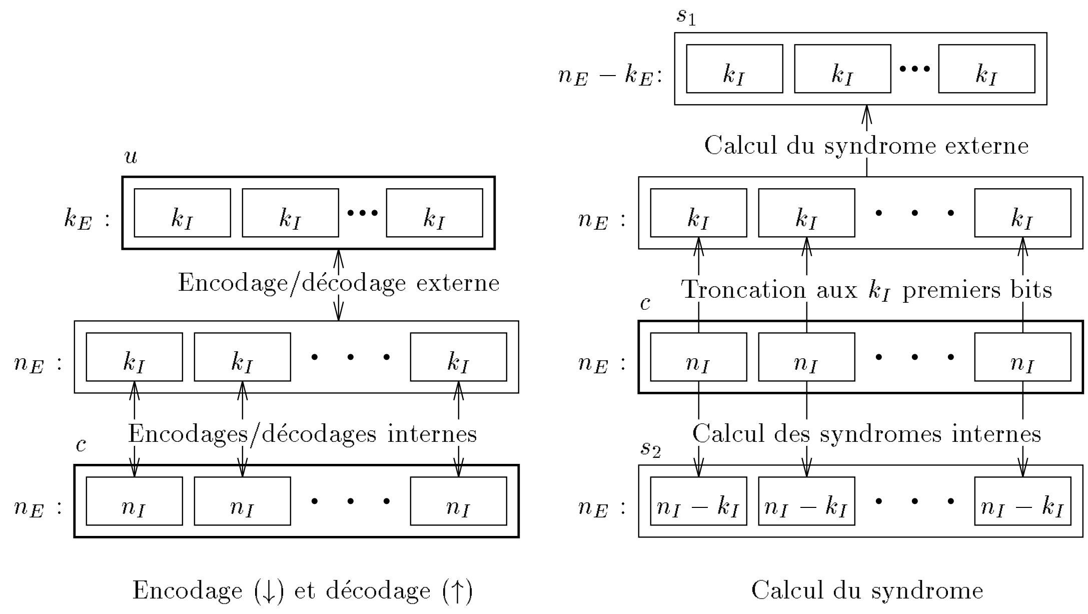

FIGURE 3.3. Opérations dans un code concaténé

1. Il faut commencer par découper le mot c en petits vecteurs de n_I bits. De là, le travail se fait en 2 parties, comme le montre la figure.
2. D'un côté, nous fabriquons un mot comprenant n_E morceaux de k_I bits, en tronquant chaque morceau de n_I bits. Nous pouvons directement transformer ce mot en un polynôme sur le corps fini et en calculer le syndrome selon le code externe. En ramenant ce résultat en bits, nous avons un mot de $(n_E - k_E)k_I$ bits, s_1. Nous venons de calculer la partie du haut de la figure.
3. Revenons au mot original. Si nous considérons chaque morceau de n_I bits comme un mot dans le code interne, nous pouvons calculer le syndrome de chacun d'eux. En alignant ces syndromes, nous obtenons un mot de $n_E(n_I - k_I)$ bits, s_2.

Si on concatène s_1 et s_2, nous obtenons un mot de $n_E n_I - k_E k_I$ bits, ce qui correspond à la longueur prévue du syndrome. Ceci est véritablement un syndrome, car:

- C'est une fonction linéaire, puisque c'est une concaténation de résultats de fonctions linéaires.
- Si c est un mot de code, le syndrome externe s_1 sera égal à 0, de même que chacun des syndromes internes formant s_2. Le résultat final est donc le mot $\vec{0}$.
- Si c n'est pas un mot du code, soit le syndrome externe s_1, soit un des syndromes internes formant s_2, sera non nul, ce qui donnera un mot non nul comme résultat final.

Le décodage se fait à l'inverse de l'encodage (partie gauche de la figure 3.3, de bas en haut):

1. Nous commençons par découper le mot c en petits vecteurs de n_I bits.
2. Nous décodons ces petits morceaux selon le code interne. Nous avons maintenant n_E morceaux de k_I bits chacun, ce qui forme un mot pour le code externe.
3. Nous retournons le résultat du décodage selon le code externe.

Étrangement, dans le cas des codes concaténés, il est plus simple de décoder un mot que de le corriger. En fait, pour corriger un mot, on le décode, puis on l'encode de nouveau.

5. Fonctions de hachage

Une fonction de hachage, dans sa plus simple expression, est simplement une fonction $f : A \to B$, où $|A| > |B|$, servant à faire de l'adressage dispersé. Malheureusement, à chaque fonction de hachage correspond au moins un ensemble qu'elle disperse très mal. C'est pourquoi nous allons considérer des ensembles de fonctions de hachage et en choisir une au hasard dans cet ensemble, dans le but de réduire les chances que l'ensemble à disperser A soit un mauvais ensemble pour la fonction choisie. Intuitivement, nous voudrions que pour tout ensemble de données fixé, il y ait le moins possible de fonctions de hachage de l'ensemble qui se *comportent mal*. Cette notion est floue. Une façon de la formaliser a été proposée par Carter et Wegman. Soit H, une classe de fonctions de hachage de A vers B. La classe H est dite *universelle*$_2$ si pour toute paire x et y d'éléments distincts de A,

$$\#\{h \in H | h(x) = h(y)\} \leq |H|/|B|.$$

Dans **QSim**, deux genres de fonctions de hachage sont implantés: les fonctions de hachage numériques (dont le domaine et l'image sont des ensembles d'entiers) et les fonctions de hachage binaires (dont le domaine et l'image sont des ensembles de vecteurs de bits). Tous les types de fonctions de hachage, sauf un, sont des classes de fonctions de hachage universelle$_2$.

5.1. Fonctions de hachage numériques. Les fonctions de hachage numériques viennent en 2 saveurs, soit `DivHashing` et `H1Hashing`. Ces 2 types offrent la même interface d'utilisation:

- `()` (Évaluation en un point du domaine)
- `a` (Taille du domaine)
- `b` (Taille de l'image)
- `delta` (Nombre de valeurs inverses associées à une valeur dans l'image)
- `inverse` (Accès aux valeurs inverses associées à une valeur dans l'image)

5.1.1. *Type* `DivHashing`. C'est ce type de fonction de hachage qui n'est pas une classe universelle$_2$. Une fonction de ce type est définie par deux paramètres:

- a, la taille du domaine
- d, un paramètre de division

Pour une telle fonction h, $A = \{0, 1, \ldots, a-1\}$, $B = \{0, 1, \ldots, \lfloor (a-1)/d \rfloor\}$ et, pour tout $x \in A$, $h(x) = \lfloor x/d \rfloor$. Ce type de fonction est utilisé pour découper un ensemble d'entiers en blocs consécutifs de longueur égale. Seul le dernier bloc n'est pas toujours de la même longueur que les autres.

5.1.2. *Type* `H1Hashing`. Ce type implante la classe de fonctions de hachage universelle$_2$ H_1. Une fonction de cette classe est définie par quatre paramètres:

- a, la taille du domaine. De a, nous calculons p, le plus petit nombre premier supérieur ou égal à a, par une recherche séquentielle avec test de primalité naïf.
- b, la taille de l'image
- m, choisi entre 1 et $p-1$
- n, choisi entre 0 et $p-1$

Pour une telle fonction h, $A = \{0, 1, \ldots, a-1\}$, $B = \{0, 1, \ldots, b-1\}$ et, pour tout $x \in A$, $h(x) = ((mx + n) \mod p) \mod b$.

5.1.3. *Partition de bits.* Il est possible de combiner une fonction de hachage numérique et un vecteur de bits (en autant que la longueur du vecteur soit égale à la taille du domaine de la fonction). Le résultat de cette combinaison est une partition du vecteur de bits. En effet, le vecteur de bits est partitionné en b parties, une partie pour chaque valeur de l'image. Les bits d'une partie i sont les bits du vecteur aux positions x telles que $h(x) = i$. Ceci est réellement une partition, car toutes les positions sont couvertes. Ce type de structure est appelé `PartitionBit`. Les opérations usuelles sur ce type sont:

- `part_index` (Numéro de la partie où se trouve une position)
- `bit_index` (Position dans le vecteur correspondant à une position dans une partie)
- `part_length` (Taille d'une partie)
- `parity` (Parité d'une partie)
- `parities` (Vecteur des parités de toutes les parties)
- `()` (Accès d'un bit via une position dans une partie)

5.2. Fonctions de hachage binaires. Plutôt que de vouloir hacher des valeurs entières, il est souvent plus pratique de hacher des vecteurs de bits de grande taille dans des vecteurs de bits de plus petite taille. Une fonction de hachage de ce genre est définie par $h : \{0,1\}^n \to \{0,1\}^r$. Les opérations usuelles sur ces fonctions de hachage sont:

- `n` (Dimension des vecteurs dans le domaine)
- `r` (Dimension des vecteurs dans l'image)
- `()` (Évaluation en un point du domaine)

Pour l'instant, une seule classe de ces fonctions de hachage est implantée: c'est la classe universelle$_2$ H_3. Une telle fonction est définie par trois paramètres:

- n, la dimension des vecteurs dans le domaine
- r, la dimension des vecteurs dans l'image
- M, une matrice binaire de dimension $r \times n$

Pour une telle fonction h, $A = \{0,1\}^n$, $B = \{0,1\}^r$ et, pour tout $x \in A$, $h(x) = Mx$.

6. Communication

La simulation de protocoles interactifs implique une utilisation fréquente par un client de canaux de communication soit directement avec le serveur, soit avec d'autres clients. Pour chacun de ces genres de communications, une structure de données est disponible pour en faciliter l'utilisation.

6.1. Type `Client`. C'est un objet de ce type qui représente le canal reliant le client au serveur. C'est grâce à un objet `Client` que le client peut envoyer une commande au serveur. Ces commandes sont:

- `login` (Début d'une session avec le serveur)
- `link` (Établissement d'une liaison avec un autre client)
- `logout` (Fin de session)
- `kill_server` (Terminaison du serveur)
- `emit_qphoton` (Émission d'un photon de polarisation aléatoire)
- `emit_simple` (Émission d'un photon de polarisation 0)
- `emit_EPR` (Émission d'une paire de particules EPR)
- `apply_qoperator` (Application d'un opérateur unitaire sur un groupe de photons)
- `apply_qmeasure` (Application d'une mesure quantique sur un groupe de photons)
- `absorb_qphoton` (Absorption d'un photon pour qu'il puisse être émis de nouveau)

Normalement, lorsque le programme client est exécuté, le `login` a déjà été fait selon les informations fournies par l'usager. Notez que la commande `kill_server` n'est accessible qu'à l'usager `root`.

6.2. Type `Party`. D'un autre côté, le type `Party` a été créé pour faciliter les communications entre clients. Pour que 2 clients A et B puissent communiquer entre eux, il faut que A demande au serveur de créer une liaison avec B avec la commande `link` du serveur et que B fasse de même pour A. Cette commande retourne un objet de type `Party` qui servira de *poignée* pour toute communication avec l'autre participant. Les opérations usuelles sur ce type de canal sont:

- `<<` (Envoi de données à un autre client)
- `>>` (Réception de données venant d'un autre client)

Les 2 opérations `<<` et `>>` permettent la transmission de n'importe quel (ou presque...) type de donnée. La simulation rapide du canal quantique utilisent aussi cette notation. La source du canal quantique utilisera une commande du genre:

```
partenaire1 << QCUse( bases, bits );
```

alors que le récepteur utilisera la commande complémentaire:

```
partenaire2 >> QCUse( bases, bits );
```

La source fournit 2 vecteurs de bits en entrée, un pour les bases et un pour les bits d'information à envoyer. Le simulateur s'occupe d'émettre les photons correspondants et de les envoyer au récepteur. Le récepteur fournit le vecteur des bases et reçoit le vecteur des bits correspondant aux photons reçus mesurés avec ces bases.

7. Références

Les codes de Hamming furent découverts par Golay [17] et Hamming [18].

Les codes de Reed-Solomon furent cités pour la première fois en tant que codes correcteurs dans [22].

Les codes concaténés furent introduits pour la première fois par Forney [16]. Ce livre discute aussi de façon très poussée des codes de Reed-Solomon.

L'algorithme de Berlekamp pour le décodage des codes alternants est présenté dans [7].

Pour toute autre information concernant les codes correcteurs d'erreurs, veuillez consulter la bible des codes correcteurs [20].

La notion de classes de fonctions de hachage universelles$_2$, de même que la définition des classes H_1 et H_3, furent introduites par Carter et Wegman dans [13]. Dans [28], ils approfondissent la question des classes universelles$_2$ en introduisant les classes fortement universelles$_2$.

CHAPITRE 4

UTILISATION DU SIMULATEUR

1. Introduction

Dans les chapitres précédents, nous avons parlé de simulation quantique et de structures de données. Le reste, c'est vous qui le faites. C'est le *credo* de **QSim**. Vous êtes libre de combiner à votre guise ces concepts et de construire un programme simulant un protocole de votre cru. Ceci est le principal objectif de **QSim**. Il offre un grand nombre de primitives faciles à utiliser. Ces primitives ne sont pas à vocation strictement cryptographique. Vous n'êtes même pas obligés de simuler un protocole. À la limite, vous pourriez vous en servir pour échanger du texte ASCII sur Internet...

Il vous serait possible d'utiliser les primitives de simulation quantique pour toutes sortes d'application:

- Interaction de plusieurs états enchevêtrés
- Simulation à petite échelle d'un ordinateur quantique
- Téléportation de plus d'un état à la fois
- Implantation d'attaques au niveau quantique sur les protocoles décrits plus loin

Nous allons maintenant présenter 4 protocoles prédéfinis utilisant les primitives de simulation quantique. Il s'agit de:

- **QBC**, la mise en gage quantique
- **QOT**, le transfert inconscient quantique
- **QKD**, l'échange de clés quantique
- **QT**, la téléportation quantique

Rappellons-nous que ces protocoles sont une couche ajoutée par-dessus les structures de données et les primitives quantiques. Ils sont présents pour éviter d'avoir à les reprogrammer sans arrêt et aussi pour donner une idée de ce qu'il est possible de faire avec **QSim**.

Pour chacun de ces protocoles, nous discuterons de son historique, présenterons son utilité et décrirons chacune de ses parties sous forme d'un algorithme décrit en langage courant. Nous ferons souvent mention de concepts vus dans les chapitres précédents.

2. **QBC**, LA MISE EN GAGE QUANTIQUE

Un protocole de mise en gage de bit permet à *Alice* d'envoyer un bit à *Bob*, tel que ce bit est camouflé aux yeux de *Bob* et n'est pas modifiable à distance par *Alice*. *Alice* peut ensuite révéler la valeur du bit à *Bob* au moment qu'il lui plaira. De façon imagée, c'est comme si *Alice* place son bit dans un coffre-fort qu'elle seule peut ouvrir, et qu'elle donne le coffre-fort fermé à *Bob*. Elle révèle le bit en lui donnant la combinaison du coffre. L'implantation de ce protocole dans **QSim** s'exécute en 3 parties:

1. Initialisation
2. Camouflage du bit en un pâté[1]
3. Révélation du bit et validation du pâté

L'initialisation fixe certaines valeurs qui ne changent pas souvent lorsque le protocole est exécuté plusieurs fois de suite, comme par exemple le taux d'erreur à la réception du canal quantique et le paramètre de sécurité. Dans le pseudo-code qui suit, *Alice* est la source du canal quantique et *Bob* est le récepteur. Pour une implantation où les rôles sont inversés, voir l'Annexe A.

1. *Bob* fixe son taux d'erreur à la réception ϵ et choisit un paramètre de sécurité N.
2. *Bob* choisit un code binaire aléatoire C, de longueur N et de dimension $0.52N$.
3. *Bob* envoie ϵ, N et C à *Alice*.
4. *Alice* vérifie que C a les bons paramètres et que ϵ est inférieur à 1%[2].
5. *Alice* choisit un vecteur aléatoire de N bits r et l'envoie à *Bob*.

Dans l'étape du camouflage, *Alice* possède un bit x qu'elle veut envoyer à *Bob* sous la forme camouflée d'un pâté. *Bob* conservera ce pâté en attendant qu'*Alice* lui donne les moyens de l'ouvrir. Le camouflage est une opération efficace, car elle ne comporte que quelques choix aléatoires et un transfert quantique de N photons. Il est souhaitable que cette opération soit efficace car il arrive souvent que, dans un protocole, nous camouflions un grand nombre de bits, mais que nous n'en révélions que la moitié, parfois même beaucoup moins.

1. *Alice* choisit un vecteur de N bases aléatoires B.
2. *Bob* choisit un vecteur de N bases aléatoires B'.

[1] Cette appelation est une contribution de M. Claude Goutier.
[2] La preuve de la sécurité du protocole dépend de cette limite.

3. *Alice* choisit au hasard un mot du code c, tel que $c \odot r = x$.
4. *Alice* transmet c sur le canal quantique avec ses bases B. *Bob* utilise ses bases B' pour recevoir du canal quantique un mot c'.

Dans l'étape de révélation, *Alice* révéle à *Bob* toute l'information qu'elle possède, y compris la valeur du bit x. *Bob* utilise cette information pour vérifier la validité du pâté qu'il a reçu d'*Alice*.

1. *Alice* envoie B, c et x à *Bob*.
2. *Bob* vérifie que
   - le nombre de positions i telles que $B_i = B'_i$ mais $c_i \neq c'_i$ est inférieur à $0.07N$,
   - c est un mot du code C,
   - $c \odot r = x$.

   Si une de ces vérifications échoue, alors *Bob* rejette, sinon il accepte.

3. **QOT**, LE TRANSFERT INCONSCIENT QUANTIQUE

Le transfert inconscient est une primitive souvent utilisée dans les protocoles cryptographiques. Dans ce protocole, *Alice* possède 2 bits, b_0 et b_1, et veut permettre à *Bob* d'apprendre la valeur de l'un d'entre eux. *Bob* décide de l'indice c du bit qu'il veut connaître et *Alice* ne doit pas pouvoir apprendre quoi que ce soit sur cet indice. De plus, *Bob* ne doit pouvoir obtenir d'information que sur le bit b_c. L'implantation de ce protocole dans **QSim** s'exécute en 2 parties:

1. Initialisation
2. Exécution principale

Comme pour le protocole précédent, l'initialisation fixe certaines valeurs qui ne changent pas souvent lorsque le protocole est exécuté plusieurs fois de suite. Dans le pseudo-code qui suit, *Alice* est la source du canal quantique et *Bob* est le récepteur. Pour une implantation où les rôles sont inversés, voir l'Annexe A.

1. *Bob* fixe son taux d'erreur à la réception ϵ et choisit un paramètre de sécurité S.
2. *Bob* trouve un code concaténé Reed-Solomon/Hamming C, de paramètres (N, K, D), tel que $N \geq S$, $K > 0.51N$ et $D \geq 2\epsilon N + 1$. Nous avons choisi un code de Hamming comme code interne car ces codes sont plus efficaces, pour de petites valeurs de N (en deça de 1000000), que les codes binaires aléatoires. Les valeurs de N possibles pour cette implantation vont de 42 à 2080374753.
3. *Bob* envoie ϵ, S et C à *Alice*.
4. *Alice* vérifie que les paramètres du code satisfont les conditions énoncées en 2.

La prochaine étape constitue le cœur du protocole. Après plusieurs échanges de données, *Alice* arrive à produire 2 valeurs dont une seule est utilisable par *Bob*. Cette valeur permettra à *Bob* de calculer la valeur du bit qu'il désire connaître.

1. *Alice* choisit un vecteur de $2N$ bits aléatoires u et un vecteur de $2N$ bases aléatoires B.
2. *Bob* choisit un vecteur de $2N$ bases aléatoires B'.
3. *Alice* envoie u sur le canal quantique avec ses bases B. *Bob* utilise ses bases B' pour recevoir du canal quantique le vecteur u'.
4. *Alice* envoie ses bases B à *Bob*.
5. *Bob* sépare l'ensemble d'indices $\{0, 1, \ldots, 2N-1\}$ en 2 sous-ensembles disjoints I_0 et I_1 de N éléments chacun. Dans I_0, nous plaçons le maximum d'indices i tels que $B_i = B'_i$ et nous plaçons le reste dans I_1.
6. *Bob* génère un bit aléatoire $\hat{c}$.
7. *Bob* envoie $I_{\hat{c}}$ et $I_{1-\hat{c}}$ à *Alice*, dans cet ordre, qu'elle reçoit comme étant J_0 et J_1.
8. *Alice* construit u_0 en concaténant les bits de u aux positions décrites dans J_0, et u_1 de la même façon avec J_1.
9. *Alice* calcule les syndromes de u_0 et de u_1, respectivement s_0 et s_1, et les envoie à *Bob*.
10. *Bob* construit v à partir des bits de u' aux positions décrites dans I_0.
11. *Bob* génère un mot aléatoire de syndrome $s_{\hat{c}}$, soit w.
12. *Bob* corrige le mot $v \oplus w$ selon le code C, puis en calcule le ou-exclusif avec w, pour obtenir x, une version corrigée de v.
13. *Alice* envoie à *Bob* 2 vecteurs de N bits aléatoires, p_0 et p_1.
14. *Bob* dit à *Alice* si $\hat{c} = c$ ou non.
15. *Alice* calcule les produits scalaires $d_0 = p_0 \odot u_0$ et $d_1 = p_1 \odot u_1$. Si $\hat{c} = c$, *Alice* envoie à *Bob* les valeurs $d_0 \oplus b_0$ et $d_1 \oplus b_1$. Sinon, elle envoie à *Bob* les valeurs $d_0 \oplus b_1$ et $d_1 \oplus b_0$. *Bob* reçoit ces valeurs comme étant d'_0 et d'_1.
16. *Bob* obtient finalement $b_c = d'_{\hat{c}} \oplus (x \odot p_{\hat{c}})$.

4. **QKD**, L'ÉCHANGE DE CLÉS QUANTIQUE

L'échange de clés est un protocole entre 2 partenaires coopératifs. À la fin du protocole, les deux partenaires partagent une clé secrète, et tout espion observant le protocole en action ne peut posséder sur cette clé que très peu d'information. Le protocole classique de Diffie et Hellman est un exemple bien connu de protocole d'échange de clés. L'échange de clés quantique fut conçu par Bennett et Brassard. Au point de vue de la sécurité, le protocole de Diffie et

Hellman est au mieux calculatoirement sûr, alors que le protocole de Bennett et Brassard est inconditionnellement sûr. L'implantation de ce protocole dans **QSim** s'exécute en 2 étapes:

1. Initialisation
2. Exécution principale

Tout comme pour les 2 protocoles précédents, l'initialisation fixe certaines valeurs qui ne changent pas souvent lorsque le protocole est exécuté plusieurs fois de suite. Dans le pseudo-code qui suit, *Alice* est la source du canal quantique et *Bob* est le récepteur. Contrairement aux 2 protocoles précédents, **QKD** est symétrique quant à son résultat, car les 2 partenaires obtiennent la même chaîne de bits à la fin. Pour fins de simplicité, nous supposerons que le participant qui envoie les photons dans le canal quantique jouera aussi le rôle d'*Alice* dans le protocole, alors que l'autre jouera celui de *Bob*.

1. *Bob* fixe son taux d'erreur à la réception ϵ et choisit un paramètre de sécurité S. ϵ est le taux d'erreur résultant d'une utilisation *honnête* du canal quantique (Voir remarque à la fin de la section).
2. *Bob* envoie ϵ et S à *Alice*.

La prochaine étape constitue le cœur du protocole. Premièrement, *Alice* et *Bob*, en utilisant le canal quantique sur un nombre N de bits, en arrivent à partager une chaîne d'environ $\frac{N}{2}$ bits, qui diffère en plusieurs endroits. Pour réconcilier leurs chaînes de bits, ils utilisent le protocole de réconciliation Cascade, une technique très efficace décrite dans l'Annexe A. Leurs chaînes sont maintenant identiques, mais les échanges requis par le protocole Cascade a pu livrer un certain nombre de bits à un espion. Un protocole de distillation de secrets sera finalement appliqué, ce qui réduira à presque rien l'information possédée par cet espion. Voici une description plus détaillée de ces actions:

1. *Alice* choisit un vecteur de N bits aléatoires u et un vecteur de N bases aléatoires B.
2. *Bob* choisit un vecteur de N bases aléatoires B'.
3. *Alice* envoie u sur le canal quantique avec ses bases B. *Bob* utilise ses bases B' pour recevoir du canal quantique le vecteur u'.
4. *Bob* envoie ses bases B' à *Alice*.
5. *Alice* indique à *Bob* les positions où les bases coïncident.
6. *Alice* ne garde que les bits de u où les bases coïncident pour obtenir le vecteur $\hat{u}$.
7. *Bob* fait de même avec u' pour obtenir le vecteur $\hat{u}'$.
8. *Alice* et *Bob* exécutent le protocole Cascade pour réconcilier leurs chaînes de bits respectives, $\hat{u}$ et $\hat{u}'$. Leurs chaînes devraient maintenant être identiques, au prix d'avoir livré à un espion un certain nombre r de bits d'information.

9. Soit l, la longueur de $\hat{u}$ et de $\hat{u}'$. *Alice* choisit aléatoirement une fonction de hachage h dans la classe universelle$_2$ H_3, telle que $h : \{0,1\}^l \rightarrow \{0,1\}^{l-r-t-S}$ et l'envoie à *Bob*. Cette classe est décrite dans le précédent chapitre. Le paramètre t représente le nombre de bits estimé qu'un espion a pu obtenir avant le début de la réconciliation. La valeur de ce paramètre depend du nombre d'erreurs observé. Si l'espion est limité à des mesures individuelles sur les photons, une formule décrite dans [2] donne une borne supérieure. Dans le cas général, on ne connaît pas encore de méthode pour calculer cette valeur.
10. *Alice* et *Bob* appliquent la fonction de hachage sur leurs chaînes de bits respectives. La chaîne de $l - r - S - t$ bits résultante constituera la clé secrète partagée.

Notre implantation de ce protocole diffère du protocole original de Bennett et Brassard, car nous ne considérons aucune fraude au niveau du canal quantique (donc $t = 0$). Les seules informations apprises par un espion le sont au niveau du protocole Cascade. Dans ce cas précis, la distillation de secrets n'est pas la meilleure façon de réduire l'information qu'un espion a pu apprendre en écoutant le canal public. C'est toutefois ce que nous avons implanté dans le but d'être le plus général possible.

5. **QT**, LA TÉLÉPORTATION QUANTIQUE

La téléportation dont on parle dans cette section est encore très loin des miracles de la science-fiction. Nous ne téléportons pour l'instant que des états quantiques, comme par exemple la polarisation d'un photon. Le processus implique 3 particules. La téléportation consiste à transformer une particule *cible* pour que sa polarisation soit la même que celle d'une particule *source*. Une troisième particule sert de catalyseur, en quelque sorte. Un effet inévitable de cette transformation est que la particule source est elle-aussi transformée dans le processus. C'est donc une *copie*, avec *destruction* de l'original.

Le protocole de téléportation est légèrement différent des protocoles précédents. En effet, il n'utilise pas à proprement parler le canal quantique. Cependant, il utilise les primitives quantiques offertes par **QSim** présentées au Chapitre 2. Notre implantation permet de téléporter un photon, désigné par son étiquette, d'un usager à un autre. L'implantation de ce protocole dans **QSim** s'exécute en 2 étapes:

1. Partage d'une paire E-P-R
2. Téléportation d'une particule

Le partage d'une paire E-P-R est vraiment très simple. Cependant, pour chaque téléportation, il faut qu'il y ait eu partage d'une paire E-P-R au préalable. Il est possible de partager un grand nombre de paires E-P-R, puis d'opérer une séquence de téléportations.

1. *Alice* crée une paire E-P-R, p_1 et p_2.
2. *Alice* envoie une des 2 particules à *Bob*, soit p_2.

La téléportation s'opère en appliquant une mesure conjointe sur la particule à téléporter et une des particules E-P-R, suivie d'un opérateur de correction sur l'autre particule E-P-R. Cette mesure est la mesure de Bell, une mesure projetant un état à 2 particules sur un des vecteurs de la base de Bell

$$\left\{\left(0, \frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0\right), \left(0, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right), \left(\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\right), \left(\frac{1}{\sqrt{2}}, 0, 0, -\frac{1}{\sqrt{2}}\right)\right\}.$$

Soit q, la particule qu'*Alice* veut téléporter à *Bob*.

1. *Alice* applique la mesure de Bell conjointement sur les particules p_1 et q, et observe le résultat r, un chiffre de 0 à 3.
2. *Alice* annonce à *Bob* la valeur de r.
3. *Bob* applique à la particule p_2 un opérateur unitaire choisi en fonction de r:

$$\begin{array}{cccc} r=0 & r=1 & r=2 & r=3 \\ \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} & \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{array}$$

La particule p_2 devrait maintenant être une copie conforme de la particule q, tandis que p_1 et q ont été projetées sur un des vecteurs de la base de Bell.

6. Références

Le protocole **QBC** fut publié pour la première fois dans [9], puis amélioré dans [10].

Le premier ancêtre du transfert inconscient, le protocole de "Quantum Multiplexing" de Wiesner (en 1970), fut publié pour la première fois dans [29]. Plus tard, Rabin introduit la notion de transfert équivoque dans [21]. Le protocole **QOT** fut publié pour la première fois dans [5].

Le protocole d'échange de clés de Diffie et Hellman fut publié dans [14]. Une version quantique de l'échange de clés fut présentée pour la première fois dans [2]. Un prototype fut ensuite contruit et décrit dans [1]. Le protocole de réconciliation Cascade fut publié dans [11]. Le protocole de distillation de secrets employé à l'étape 10 du protocole **QKD** fut présenté dans [4].

Le protocole **QT** fut publié dans [3].

CONCLUSION

QSim est un projet exploratoire. Son but est principalement didactique, c'est-à-dire qu'il a été conçu pour démystifier le canal quantique et les protocoles cryptographiques, en général. Dans son état actuel, il n'offre que peu d'intérêt pour des applications industrielles.

D'un autre côté, il a rempli ses objectifs didactiques envers son créateur. Sa conception fut remplie de défis à relever. La simulation adéquate des particules quantiques en fut un qui obtint une attention particulière. En effet, elle requiert plusieurs primitives mathématiques, de même qu'une grande précision numérique. Plusieurs approches ont été tentées:

- Tout programmer en MAPLE. La motivation principale de cette approche était de profiter des primitives de calcul symbolique offertes par MAPLE. Ceci s'est révélé rapidement difficile, étant donnée la lourdeur de ce langage.
- Utiliser MAPLE comme calculatrice à partir d'un programme C. Cette approche eut pour effet d'alléger le traitement non-mathématique.
- Tout faire en C++. Nous nous sommes finalement rendus compte que la précision des nombres à point flottant du C++ était suffisante pour nos besoins.

Voici quelques extensions dont pourrait profiter **QSim**:

- Toutes les implantations actuelles de protocoles sont honnêtes. Il serait intéressant d'implanter quelques attaques sur ces protocoles. De plus, certains protocoles quantiques n'ont pas été implantés, comme par exemple l'identification quantique.
- Certaines structures de données ne sont pas implantées de façon optimale, en particulier les polynômes.
- Pour une portabilité accrue, il pourrait être envisagé de convertir les programmes de C++ vers C.

Nous espérons finalement que le travail qui a été accompli dans ce mémoire pourra être réutilisé dans une véritable implantation du canal quantique.

APPENDICE A

PROTOCOLES ALTERNATIFS

1. Introduction

Cet annexe contient quelques définitions supplémentaires de protocoles cryptographiques. Les 2 premières sections détaillent une implantation alternative des protocoles **QBC** et **QOT** lorsque *Bob* est la source du canal quantique et *Alice* est le récepteur. La dernière section contient une description du protocole de réconciliation Cascade, utilisé dans le protocole **QKD**.

2. QBC

Supposons que *Bob* soit la source du canal quantique. L'initialisation du protocole se déroule avec les rôles inversés, c'est-à-dire que *Alice* joue le rôle de *Bob*, et *Bob* celui d'*Alice*. La partie camouflage subit une transformation plus frappante.

1. *Alice* choisit un vecteur de N bases aléatoires, B.
2. *Bob* choisit un vecteur de N bases aléatoires, B'.
3. *Bob* choisit au hasard un mot u' qu'il transmet sur le canal quantique avec ses bases B'. *Alice* utilise ses bases B pour recevoir du canal quantique un mot u.
4. *Alice* envoie à *Bob* le syndrome s du mot u et la valeur de $y = (u \odot r) \oplus x$.

La partie révélation est elle aussi légèrement différente.

1. *Alice* envoie B, u et x à *Bob*.
2. *Bob* vérifie que
   - le nombre de positions i telles que $B_i = B'_i$ mais $u_i \neq u'_i$ est inférieur à $0.07N$,
   - le syndrome de u est bien s,
   - $(u \odot r) \oplus x = y$.

   Si une de ces vérifications échoue, alors *Bob* rejette, sinon il accepte.

3. QOT

Supposons que *Bob* soit la source du canal quantique. L'initialisation du protocole se déroule avec les rôles inversés, c'est-à-dire qu'*Alice* joue le rôle de *Bob* et *Bob*, celui d'*Alice*. Le début du protocole principal subit une légère modification.

1. *Alice* choisit un vecteur de $2N$ bases aléatoires B.
2. *Bob* choisit un vecteur de $2N$ bits aléatoires u' et un vecteur de $2N$ bases aléatoires B'.
3. *Bob* envoie u' sur le canal quantique avec ses bases B'. *Alice* utilise ses bases B pour recevoir du canal quantique le vecteur u.
4. *Alice* envoie ses bases B à *Bob*.

Le reste du protocole se déroule comme dans le cas normal.

4. Cascade

Le protocole Cascade s'applique dans les situations où 2 partenaires partagent une chaîne de N bits, mais dont plusieurs de ces bits sont différents, suite à des erreurs de transmission ou à l'intervention d'un espion. Il fonctionne pour des taux d'erreur inférieurs à 15%. Donc, soit u et u', les chaînes de N bits d'*Alice* et de *Bob*, respectivement.

1. *Alice* construit une permutation aléatoire à appliquer sur les bits de u, soit σ.
2. *Alice* envoie σ à *Bob*.
3. *Alice* applique la permutation à u pour obtenir v, tandis que *Bob* applique la permutation à u' pour obtenir v'.
4. *Alice* envoie à *Bob* un échantillon (de longueur constante) du vecteur v afin qu'il puisse vérifier que le nombre d'erreurs correspond au taux d'erreur prévu. Ces bits sont retirés des vecteurs v et v'. La longueur résultante de ces vecteurs sera notée l.
5. *Alice* et *Bob*, chacun de leur côté, calculent une longueur initiale de bloc, $k_1 = 0.73/\epsilon$. Cependant, si $k_1 < 5$, alors nous fixerons sa valeur à 5. De même, si $k_1 > \frac{1}{16}l$, alors nous fixerons sa valeur à $\frac{1}{16}l$. Ceci implique que l doit être supérieur à 80.
6. *Alice* découpe son vecteur de bits en blocs de k_1 bits consécutifs et calcule la parité de chacun de ces blocs. Elle envoie ces parités à *Bob*.
7. *Bob* fait le même calcul et compare les parités qu'il a calculées avec celles qu'il a reçues d'*Alice*. Pour chaque parité différente, ils utiliseront le protocole de fouille binaire interactive (décrit plus loin) pour trouver l'erreur. Quand ils ont passé à travers toutes les parités, ils signalent la fin de la ronde 1.

8. *Alice* choisit ensuite une partition aléatoire des bits de v, telle que chaque sous-ensemble de la partition a exactement $2k_1$ bits, sauf pour quelques exceptions. *Alice* envoie cette partition à *Bob*.
9. *Alice* calcule les parités de chacun de ces *sous-vecteurs* de bits et les envoie à *Bob*.
10. *Bob* fait le même calcul et compare les parités qu'il a calculées avec celles qu'il a reçues d'*Alice*. Pour chaque parité différente, ils utiliseront le protocole de fouille binaire interactive pour trouver l'erreur. Soit i, la position de ce bit dans le vecteur. À la fin de la ronde 1, toutes les parités étaient correctes. En corrigeant le bit à la position i, la parité de la ronde 1 où se trouve i n'est plus bonne. Ils utilisent le protocole de fouille binaire interactive pour corriger cette nouvelle erreur. Soit j, la position de ce bit dans le vecteur. Si ce bit se trouve dans une des parités en erreur, la cascade s'arrête ici. Sinon, elle repart de plus belle. Lorsque toutes les parités en erreur sont corrigées, les deux partenaires signalent la fin de la ronde 2.
11. *Alice* et *Bob* refont le même processus pour la ronde 3 avec une partition choisie aléatoirement dont les sous-ensembles ont presque tous $4k_1$ bits. Encore ici, chaque erreur découverte entraînera une *cascade* d'erreurs dans les sous-ensembles correspondants aux rondes 1, 2 et 3. À la fin de la ronde 3, il faut que toutes les parités des rondes 1, 2 et 3 soient identiques des 2 côtés.
12. *Alice* et *Bob* refont le même processus avec $8k_1$ pour la ronde 4.
13. *Alice* et *Bob* refont le même processus avec $16k_1$ pour la ronde 5.

Les partitions aléatoires utilisées plus haut sont implantées par le type `PartitionBit`, lui-même un mélange de `VectorBit` et de `H1Hashing`, vus au Chapitre 3. À la fin de cette étape, les 2 participants devraient posséder la même chaîne de bits. De plus, à chaque parité qu'*Alice* annonce à *Bob*, elle augmente un compteur pour connaître le nombre de bits qu'un espion aurait pu apprendre en écoutant les communications. Voici maintenant le mini-protocole de fouille binaire interactive. Soit t, le sous-vecteur en erreur d'*Alice*, et t', celui de *Bob*.

1. *Alice* calcule la parité de la moitié supérieure de t et l'envoie à *Bob*.
2. *Bob* fait le même calcul avec t'.
3. Si les parités sont différentes, les deux partenaires ne considèrent que la moitié supérieure de leur vecteur respectif et reviennent à 1.
4. Si les parités sont identiques, ils ne considèrent que la moitié inférieure de leur vecteur respectif et reviennent à 1.
5. La boucle s'arrête lorsque les vecteurs n'ont qu'un seul bit. *Bob* va finalement inverser le bit correspondant dans son vecteur original, ce qui corrigera la différence.

APPENDICE B

SOURCES C++ DES PROTOCOLES

1. Introduction

Dans cet appendice, vous trouverez les implantations de chacun des protocoles décrits dans le Chapitre 4. Ces implantations sont présentées dans leur format original, en C++. Nous espérons que cet appendice vous aidera à comprendre le fonctionnement de ces protocoles. De plus, en vous fournissant des exemples de programmation en **QSim**, cela pourrait vous aider à mieux concevoir vos propres protocoles.

2. Descriptions des classes implantant les protocoles

Voici les déclarations pour chacunes des classes de protocoles implantées.

```
/*
** Classe modelisant un pate
*/
class QBCInfo {
    Bit m_bit;              // Bit camoufle/revele
    VectorBit m_word;       // Mot envoye/recu par le canal quantique
    VectorBit m_bases;      // Bases utilises pour le transfert quantique
    VectorBit m_syndrome;   // Syndrome du mot envoye

    friend class QBC;
};

/*
** Classe modelisant le protocole QBC
*/
class QBC {
    Real m_error_rate;      // Taux d'erreur prevu du canal quantique
    VectorBit m_r;          // Vecteur r du protocole
    RandomCode m_code;      // Code binaire aleatoire
    Party *m_party;         // Partenaire dans le protocole
    Boolean m_source;       // Sommes-nous la source du canal quantique ?

  public:
    Void init_S( Party &pa );
    Void init_R( Party &pa, Real error_rate, Longint security );
    Void hide_A( const Bit &x, QBCInfo &qi );
    Void hide_B( QBCInfo &qi );
    Void reveal_A( const QBCInfo &qi );
```

```
    Void reveal_B( const QBCInfo &qi, Bit &x );
};

/*
** Classe modelisant le protocole QOT
*/
class QOT {
    Real m_error_rate;    // Taux d'erreur prevu du canal quantique
    Longint m_security;   // Parametre de securite du protocole
    ConcatCode m_code;    // Code concatene utilise pour la correction
    Party *m_party;       // Partenaire dans le protocole
    Boolean m_source;     // Sommes-nous la source du canal quantique ?

  public:
    Void init_S( Party &pa );
    Void init_R( Party &pa, Real error_rate, Longint security );
    Void run_A( const Bit &b_0, const Bit &b_1 );
    Void run_B( const Bit &c, Bit &b_c );
};

/*
** Classe modelisant le protocole QKD
*/
class QKD {
    Real m_error_rate;    // Taux d'erreur prevu du canal quantique
    Party *m_party;       // Partenaire dans le protocole
    Longint m_info;       // Nombre de bits d'information devoiles
    Longint m_security;   // Parametre de securite
    Boolean m_source;     // Sommes-nous la source du canal quantique ?

    Void cascade_A( VectorBit &key );
    Void cascade_B( VectorBit &key );
    Void binary_A( const PartitionBit &pb, Longint part );
    Longint binary_B( const PartitionBit &pb, Longint part );

  public:
    Void init_A( Party &pa );
    Void init_B( Party &pa, Real error_rate, Longint security );
    Void run_A( Longint N, VectorBit &key );
    Void run_B( Longint N, VectorBit &key );
};

/*
** Classe modelisant le protocole QT
*/
class QT {
    Party *m_party;      // Partenaire dans le protocole
    Client *m_client;    // Variable client utilisee

  public:
    Void init( Client &cl, Party &pa );
    Longint share_A();
    Longint share_B();
    Void run_A( Longint particle, Longint epr );
    Void run_B( Longint epr );
};
```

3. Implantations des fonctions dans les classes

Les 2 fonctions suivantes sont utilisées à plusieurs reprises dans les programmes qui vont suivre. Elles servent à la gestion des conditions de rejets dans les protocoles:

```
/*
** Gestion des rejets, partie active
*/
Void reject( Boolean cont, Party &pa, const Char *func ) {
    pa << cont;

    if ( !cont )
        throw (xReject( func ));
}

/*
** Gestion des rejets, partie passive
*/
Void check_reject( Party &pa, const Char *func ) {
    Boolean test;

    pa >> test;
    if ( !test )
        throw (xReject( func ));
}
```

Dans les 7 pages suivantes, vous trouverez les sources des fonctions implantant les protocoles prédéfinis. Ces sources sont en C++ et sont malheureusement avares de commentaires. Une connaissance de ce langage est fortement recommandé pour la compréhension de ces programmes. Ces programmes sont affichés dans un format *paysage* en 2 colonnes, en essayant (lorsque c'est possible) de placer les 2 fonctions complémentaires, vis-à-vis l'une de l'autre, pour chaque aspect d'un protocole. Vous pourrez ainsi suivre en parallèle l'action des 2 participants au protocole.

À la suite de ces pages de programmes, vous trouverez un exemple de session avec **QSim**. Vous verrez, en format *paysage* en 2 colonnes, ce qui est imprimé lors d'une exécution des programmes de démonstration `alice.cc` et `bob.cc`. Le protocole utilisé dans l'exemple est le protocole **QBC**.

```
/*
** QBC : Initialisation en tant que source du canal quantique
*/
Void QBC::init_S( Party &pa ) {
    m_party = &pa;
    m_source = TRUE;

    Longint security;
    pa >> m_error_rate >> security >> m_code;
    reject( m_error_rate <= 0.01 && m_code.n() == security &&
            m_code.k() == Longint(security * 0.52), pa, "QBC::init" );

    m_r.reshape( security ).randomize();

    pa << m_r;
}

/*
** QBC : Camouflage d'un bit en tant qu'Alice
*/
Void QBC::hide_A( const Bit &x, QBCInfo &qi ) {
    const Longint N = m_code.n();

    if ( m_source ) {
        qi.m_bit = x;
        qi.m_word = m_code.random_codeword();
        while ( ( m_r ^ qi.m_word ) != x )
            qi.m_word = m_code.random_codeword();

        qi.m_bases.reshape( N ).randomize();
        (*m_party) << QCUse( qi.m_bases, qi.m_word );
    }
    else {
        qi.m_bases.reshape( N ).randomize();
        (*m_party) >> QCUse( qi.m_bases, qi.m_word );
        (*m_party) << m_code.syndrome( qi.m_word )
                   << ( ( qi.m_word ^ m_r ) + x );
    }
}

/*
** QBC : Revelation du bit en tant qu'Alice
*/
Void QBC::reveal_A( const QBCInfo &qi ) {
    (*m_party) << qi.m_word << qi.m_bases << qi.m_bit;

    check_reject( *m_party, "QBC::reveal_A" );
}
```

```
/*
** QBC : Initialisation en tant que recepteur du canal quantique
*/
Void QBC::init_R( Party &pa, Real error_rate, Longint security ) {
    m_party = &pa;
    m_source = FALSE;
    m_code = RandomCode( security, Longint(0.52 * security) );

    m_error_rate = error_rate;
    pa << error_rate << security << m_code;
    check_reject( *m_party, "QBC::init" );

    pa >> m_r;
}

/*
** QBC : Camouflage d'un bit en tant que Bob
*/
Void QBC::hide_B( QBCInfo &qi ) {
    const Longint N = m_code.n();

    if ( m_source ) {
        qi.m_word.reshape( N ).randomize();
        qi.m_bases.reshape( N ).randomize();
        (*m_party) << QCUse( qi.m_bases, qi.m_word );
        (*m_party) >> qi.m_syndrome >> qi.m_bit;
    }
    else {
        qi.m_bases.reshape( N ).randomize();
        (*m_party) >> QCUse( qi.m_bases, qi.m_word );
        qi.m_bit = 0;
    }
}

/*
** QBC : Revelation du bit en tant que Bob
*/
Void QBC::reveal_B( const QBCInfo &qi, Bit &x ) {
    const Longint N = m_code.n();
    VectorBit a_word, a_bases;
    (*m_party) >> a_word >> a_bases >> x;

    Longint delta = 0;
    for ( Longint i = 0; i < N; i++ )
        if ( qi.m_bases[i] == a_bases[i] && qi.m_word[i] != a_word[i] )
            delta++;

    Boolean test_word;
    if ( m_source )
        test_word = ( m_code.syndrome( a_word ) == qi.m_syndrome );
    else
        test_word = m_code.is_codeword( a_word );

    Bit dp_bit = ( m_r ^ a_word ) + qi.m_bit;

    reject( delta < 0.07 * N && dp_bit == x && test_word,
            *m_party, "QBC::reveal_B" );
}
```

```
/*
** QOT : Initialisation en tant que source du canal quantique
*/
Void QOT::init_S( Party &pa ) {
    m_party = &pa;
    m_source = TRUE;

    pa >> m_error_rate >> m_security;

    check_reject( pa, "QOT::init" );

    Longint ki, ni, de;

    pa >> ki >> ni >> de;
    const Longint ne = ( 1L << ki ) - 1;

    reject( m_error_rate <= 0.01 && ne * ni >= m_security,
            *m_party, "QOT::init" );

    m_code.set_codes( HammingCode( ni, ki ), RSCode( ki, ne - de + 1 ) );
}
```

```
/*
** QOT : Initialisation en tant que recepteur du canal quantique
*/
Void QOT::init_R( Party &pa, Real error_rate, Longint security ) {
    m_party = &pa;
    m_source = FALSE;

    m_error_rate = error_rate;
    m_security = security;

    pa << error_rate << security;

    Longint ki = 3, ni, ne, de;
    Real nb_err;

    while ( ( ni = HammingCode::find_n( ki ) ) <= 31 ) {
        ne = ( 1L << ki ) - 1;
        nb_err = 5.0 * sqrt( ne * ni * error_rate * ( 1.0 - error_rate ) )
               + ne * ni * error_rate;
        de = 2 * Longint( 0.999 + nb_err ) + 1;
        if ( de < ne && ne * ni >= security )
            break;

        ki++;
    }

    reject( ni <= 31, pa, "QOT::init" );

    pa << ki << ni << de;
    check_reject( pa, "QOT::init" );

    m_code.set_codes( HammingCode( ni, ki ), RSCode( ki, ne - de + 1 ) );
}
```

```
/*
** QOT : Partie principale du protocole, en tant qu'Alice
*/
Void QOT::run_A( const Bit &b_0, const Bit &b_1 ) {
    const Longint N = m_code.n();
    VectorBit bases( 2 * N );
    VectorBit bits( 2 * N );
    bases.randomize();
    if ( m_source ) {
        bits.randomize();
        (*m_party) << QCUse( bases, bits );
    }
    else
        (*m_party) >> QCUse( bases, bits );

    (*m_party) << bases;

    VectorBit set_mask;
    (*m_party) >> set_mask;

    VectorBit set_0( N ), set_1( N );
    Longint ind_0 = 0, ind_1 = 0;
    for ( Longint i = 0; i < 2 * N; i++ )
        if ( set_mask[i] )
            set_1[ind_1++] = bits[i];
        else
            set_0[ind_0++] = bits[i];

    VectorBit synd_0 = m_code.syndrome( set_0 );
    VectorBit synd_1 = m_code.syndrome( set_1 );
    (*m_party) << synd_0 << synd_1;
    check_reject( *m_party, "QOT::run_A" );

    VectorBit par_0( N );
    VectorBit par_1( N );
    par_0.randomize();
    par_1.randomize();

    (*m_party) << par_0 << par_1;

    const Bit x_0 = par_0 ^ set_0;
    const Bit x_1 = par_1 ^ set_1;

    Boolean c_test;
    (*m_party) >> c_test;

    Bit bx_0, bx_1;
    if ( c_test ) {
        bx_0 = x_0 + b_0;
        bx_1 = x_1 + b_1;
    }
    else {
        bx_0 = x_0 + b_1;
        bx_1 = x_1 + b_0;
    }
    (*m_party) << bx_0 << bx_1;
}
```

```
/*
** QOT : Partie principale du protocole, en tant que Bob
*/
Void QOT::run_B( const Bit &c, Bit &b_c ) {
    const Longint N = m_code.n();
    VectorBit bases( 2 * N );
    VectorBit bits( 2 * N );

    bases.randomize();
    if ( m_source ) {
        bits.randomize();
        (*m_party) << QCUse( bases, bits );
    }
    else
        (*m_party) >> QCUse( bases, bits );

    VectorBit a_bases;
    (*m_party) >> a_bases;

    const Bit c_hat = randomBit();
    VectorBit set_mask( 2 * N, 0 );
    Longint ind_0 = 0, ind_1 = 0, i;
    VectorBit sets[2] = { N, N };

    if ( c_hat == 0 )
        for ( i = 0; i < 2 * N; i++ )
            if ( ( bases[i] == a_bases[i] || i - ind_0 >= N ) && ind_0 < N )
                sets[0][ind_0++] = bits[i];
            else {
                sets[1][ind_1++] = bits[i];
                set_mask[i].toggle();
            }
    else
        for ( i = 0; i < 2 * N; i++ )
            if ( ( bases[i] == a_bases[i] || i - ind_1 >= N ) && ind_1 < N ) {
                sets[1][ind_1++] = bits[i];
                set_mask[i].toggle();
            }
            else
                sets[0][ind_0++] = bits[i];

    (*m_party) << set_mask;

    VectorBit synd[2];
    (*m_party) >> synd[0] >> synd[1];

    const VectorBit tw = m_code.random_word( synd[c_hat] );
    Boolean ok;
    VectorBit new_set = m_code.correct( tw + sets[c_hat], ok );
    reject( ok, *m_party, "QOT::run_B" );
    new_set += tw;

    VectorBit par[2];
    (*m_party) >> par[0] >> par[1];

    const Boolean test_c_hat = ( c == c_hat );
    (*m_party) << test_c_hat;

    Bit bx[2];
    (*m_party) >> bx[0] >> bx[1];

    b_c = bx[c_hat] + ( new_set ^ par[c_hat] );
}
```

```
/*
** QKD : Initialisation en tant que source du canal quantique (Alice)
*/
Void QKD::init_A( Party &pa ) {
    m_source = TRUE;
    m_party = &pa;
    pa >> m_error_rate >> m_security;
}



/*
** QKD : Fouille binaire pour Cascade, en tant qu'Alice
*/
Void QKD::binary_A( const PartitionBit &pb, Longint part ) {
    Longint l = 0;
    Longint r = pb.part_length( part ) - 1;
    Longint m = ( l + r ) / 2;
    Boolean is_left;

    while ( l < r ) {
        (*m_party) << pb.parity( part, l, m );
        m_info++;
        (*m_party) >> is_left;

        if ( is_left )
            r = m;
        else
            l = m + 1;
        m = ( l + r ) / 2;
    }
}
```

```
/*
** QKD : Initialisation en tant que recepteur du canal quantique (Bob)
*/
Void QKD::init_B( Party &pa, Real error_rate, Longint security ) {
    m_source = FALSE;
    m_error_rate = error_rate;
    m_security = security;
    m_party = &pa;
    pa << error_rate << security;
}

/*
** QKD : Fouille binaire pour Cascade, en tant que Bob
*/
Longint QKD::binary_B( const PartitionBit &pb, Longint part ) {
    Longint l = 0;
    Longint r = pb.part_length( part ) - 1;
    Longint m = ( l + r ) / 2;
    Bit p;
    Boolean is_left;

    while ( l < r ) {
        (*m_party) >> p;
        is_left = ( pb.parity( part, l, m ) != p );
        (*m_party) << is_left;

        if ( is_left )
            r = m;
        else
            l = m + 1;
        m = ( l + r ) / 2;
    }

    return pb.bit_index( part, l );
}
```

```
/*
** QKD : Partie principale du protocole, en tant qu'Alice
*/
Void QKD::run_A( Longint N, VectorBit &key ) {
    VectorBit bases( N );
    VectorBit bits( N );

    bases.randomize();
    if ( m_source ) {
        bits.randomize();
        (*m_party) << QCUse( bases, bits );
    }
    else
        (*m_party) >> QCUse( bases, bits );

    VectorBit a_bases;
    (*m_party) >> a_bases;

    const VectorBit filter_bases = ~( bases + a_bases );
    (*m_party) << filter_bases;

    key = bits.filter( filter_bases );
    N = key.length();

    Array<Longint> perm( N );
    Longint j, t;

    for ( Longint i = 0; i < N; i++ )
        perm[i] = i;
    for ( i = N; i > 1; i-- ) {
        j = randomLongint( i );
        t = perm[i - 1];
        perm[i - 1] = perm[j];
        perm[j] = t;
    }

    (*m_party) << perm;

    VectorBit perm_key( N );
    for ( i = 0; i < N; i++ )
        perm_key[i] = key[perm[i]];

    const Longint slen = max( Longint(N * SAMPLE_RATE), 10 );

    (*m_party) << perm_key.sub( N - slen, slen );
    check_reject( *m_party, "QKD::run_A" );

    N -= slen;
    key = perm_key.sub( 0, N );

    m_info = 0;
    cascade_A( key );

    H3Hashing h3h( key.length(), key.length() - m_info - m_security );

    (*m_party) << h3h;
    key = h3h( key );
}
```

```
/*
** QKD : Partie principale du protocole, en tant que Bob
*/
Void QKD::run_B( Longint N, VectorBit &key ) {
    VectorBit bases( N );
    VectorBit bits( N );

    bases.randomize();
    if ( m_source ) {
        bits.randomize();
        (*m_party) << QCUse( bases, bits );
    }
    else
        (*m_party) >> QCUse( bases, bits );

    VectorBit filter_bases;

    (*m_party) << bases;
    (*m_party) >> filter_bases;
    key = bits.filter( filter_bases );
    N = key.length();

    Array<Longint> perm( N );
    VectorBit perm_key( N );

    (*m_party) >> perm;
    for ( Longint i = 0; i < N; i++ )
        perm_key[i] = key[perm[i]];

    const Longint slen = max( Longint(N * SAMPLE_RATE), 10 );
    const VectorBit sample = perm_key.sub( N - slen, slen );
    VectorBit a_sample;
    (*m_party) >> a_sample;
    const Longint delta = hamming( a_sample, sample );

    reject( delta < m_error_rate * 2.0 * N, *m_party, "QKD::run_B" );

    N -= slen;
    key = perm_key.sub( 0, N );

    cascade_B( key );

    H3Hashing h3h;
    (*m_party) >> h3h;
    key = h3h( key );
}
```

```
/*
** QKD : Protocole Cascade en tant qu'Alice
*/
Void QKD::cascade_A( VectorBit &key ) {
    const Longint N = key.length();
    const Longint k1 = min( max( Longint(0.73 / m_error_rate), 5 ), N / 16 );
    Longint i, b, part;
    Integer old_level;
    DivHashing dh( N, k1 );
    H1Hashing hh[4];
    PartitionBit pb[5] = { key, key, key, key, key };

    pb[0].set_hashing( &dh );
    for ( i = 1; i < 5; i++ ) {
        b = N / ( k1 << i );
        hh[i - 1].init( N, b );
        (*m_party) << hh[i - 1];
        pb[i].set_hashing( &hh[i - 1] );
    }

    for ( Integer level = 0; level < 5; level++ ) {
        VectorBit pv = pb[level].parities();
        (*m_party) << pv;
        m_info += pv.length();

        while ( 1 ) {
            (*m_party) >> part;
            if ( part == -1 )
                break;

            binary_A( pb[level], part );
        }

        while ( 1 ) {
            (*m_party) >> old_level;
            if ( old_level == -1 )
                break;

            (*m_party) >> part;
            binary_A( pb[old_level], part );
        }
    }
}
```

```
/*
** QKD : Protocole Cascade en tant que Bob
*/
Void QKD::cascade_B( VectorBit &key ) {
    const Longint N = key.length();
    const Longint k1 = min( max( Longint(0.73 / m_error_rate), 5 ), N / 16 );
    Longint i, b, x, l;
    Integer set_level;
    DivHashing dh( N, k1 );
    H1Hashing hh[4];
    PartitionBit pb[5] = { key, key, key, key, key };
    VectorBit sets[5], parities_a, parities;

    pb[0].set_hashing( &dh );
    sets[0].reshape( N / k1 ).fill( 0 );
    for ( i = 1; i < 5; i++ ) {
        b = N / ( k1 << i );
        (*m_party) >> hh[i - 1];
        pb[i].set_hashing( &hh[i - 1] );
        sets[i].reshape( b ).fill( 0 );
    }

    for ( Integer level = 0; level < 5; level++ ) {
        parities = pb[level].parities();
        (*m_party) >> parities_a;
        for ( i = 0; i < parities.length(); i++ )
            if ( parities[i] != parities_a[i] ) {
                (*m_party) << i;
                x = binary_B( pb[level], i );
                key[x].toggle();
                for ( l = 0; l < level; l++ )
                    sets[l][pb[l].part_index( x )].toggle();
            }
        (*m_party) << -1L;

        set_level = 0;
        while ( set_level <= level ) {
            i = sets[set_level].find( 1 );
            if ( i == -1 )
                set_level++;
            else {
                (*m_party) << set_level << i;
                x = binary_B( pb[set_level], i );
                key[x].toggle();
                for ( l = 0; l <= level; l++ )
                    sets[l][pb[l].part_index( x )].toggle();
                set_level = 0;
            }
        }
        (*m_party) << -1L;
    }
}
```

```
/*
** QT : Initialisation du protocole
*/
Void QT::init( Client &cl, Party &pa ) {
    m_party = &pa;
    m_client = &cl;
}

/*
** QT : Partage d'une paire de particules E-P-R, en tant qu'Alice
*/
Longint QT::share_A() {
    Longint p1, p2;

    m_client->emit_EPR( p1, p2 );
    (*m_party) << p2;

    return p1;
}

/*
** QT : Partie principale du protocole, en tant qu'Alice
*/
Void QT::run_A( Longint particle, Longint epr ) {
    Integer r = m_client->apply_qmeasure( QM_BELL, particle, epr );

    (*m_party) << r;
}
```

```
/*
** QT : Partage d'une paire de particules E-P-R, en tant que Bob
*/
Longint QT::share_B() {
    Longint p1;

    (*m_party) >> p1;

    return p1;
}


/*
** QT : Partie principale du protocole, en tant que Bob
*/
Void QT::run_B( Longint epr ) {
    Integer r;
    Longint qo_id = -1;

    (*m_party) >> r;
    switch ( r ) {
      case 0:
        qo_id = QO_SYM45; break;
      case 1:
        qo_id = QO_ROT90; break;
      case 2:
        qo_id = QO_ROT180; break;
      case 3:
        qo_id = QO_SYM0; break;
    }

    m_client->apply_qoperator( qo_id, epr );
}
```

```
Connected to the server...

Please enter your identification...
Login: alice
Password: <alice>
Login accepted...

What’s your party’s name ? bob
Trying to link with bob...done

  1) QBC (Quantum Bit Commitment)
  2) QOT (Quantum Oblivious Transfer)
  3) QKD (Quantum Key Distribution)
  4) QT (Quantum Teleportation)
Please choose a protocol to test: 1

Test QBC protocol (Alice)
1. Initialization
[Verbosity=0] Action (q,<space>,s,g,v,?): v
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  QBC initialization (source)
  1. Receive error rate, security parameter and code
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{error rate=0.01,security=100,code=Random code (100,52,?)}
  2. Test the dimension of the code
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  3. Let r be a random bit vector
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{r=[0001001001111111000011000111110110001010011111111011100100111011111000000100
0110000000101010011000000110]}
  4. Send r to party
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
2. Hide a bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{Hidden bit = 0}
  QBC hide (Alice’s part)
{bit=0}
  1. Build a random codeword
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{random codeword=[10000100110010000001110000001001110010010011001010100111110111
01000000111110011000000001011011110110100]}
  2. Send the codeword thru quantum channel
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{bases=[1001110010101000110110110010001000100100101010011010100100101110010100
111110101110100110001010110000]}
3. Reveal the bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  QBC reveal (Alice’s part)
  1. Send word, bases and bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
```

```
Connected to the server...

Please enter your identification...
Login: bob
Password: <bob>
Login accepted...

What’s your party’s name ? alice
Trying to link with alice...done

  1) QBC (Quantum Bit Commitment)
  2) QOT (Quantum Oblivious Transfer)
  3) QKD (Quantum Key Distribution)
  4) QT (Quantum Teleportation)
Please choose a protocol to test: 1

Test QBC protocol (Bob)
1. Initialization
[Verbosity=0] Action (q,<space>,s,g,v,?): v
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
Enter error rate: 0.01
Enter security parameter: 100
  QBC initialization (receptor)
  1. Creating a new random binary code
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  2. Send error rate, security parameter and code
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{error rate=0.01,security=100}
{code=Random code (100,52,?)}
  3. Receive r
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{r=[0001001001111111000011000111110110001010011111111011100100111011111000000100
0110000000101010011000000110]}
2. Hide a bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  QBC hide (Bob’s part)
  1. Receive word thru quantum channel
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{bases=[0101101001101010011110101100110000010010110110001010010100111110000000
100100110001011011110001101111]}
{word=[00000100100010001001100010000001001100001100101111010011011001000001110
11001110100010110111101100010]}
3. Reveal the bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
  QBC reveal (Bob’s part)
  1. Receive word, bases and bit
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{a word=[10000100110010000001110000001001110010010011001010100111110111010000001
11110011000000001011011110110100]}
{a_bases=[10011100101010001101101100100010001001001010100110101001001011100101
00111110101110100110001010110000]}
{bit=0}
  2. Test if Alice has cheated
[Verbosity=1] Action (q,<space>,s,g,v,?): <CR>
{delta=2}
{dp_bit=0}
{test_word=yes}
{Revealed bit = 0}
```

4. Références

Tout au long de la conception de **QSim**, c'est la bible du C++ [26] qui m'a le plus servi.

Pour la partie de **QSim** impliquant les communications UNIX et Internet entre le serveur et ses clients, le livre de Stevens [25] fut d'un grand secours.

RÉFÉRENCES

[1] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, et J. Smolin. Experimental quantum cryptography. *Journal of Cryptology*, 5(1):3–28, 1992.

[2] C. H. Bennett et G. Brassard. Quantum cryptography: Public key distribution and coin tossing. Dans *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, pages 175–179, Bangalore, India, Décembre 1984.

[3] C. H. Bennett, G. Brassard, C. Crépeau, R. Josza, A. Peres, et W. K. Wootters. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letter*, 70:1895–1899, 1993.

[4] C. H. Bennett, G. Brassard, C. Crépeau, et U. M. Maurer. Generalized privacy amplification. Dans *IEEE Transactions on Information Theory*, Novembre 1995. A paraître.

[5] C. H. Bennett, G. Brassard, C. Crépeau, et M.-H. Skubiszewska. Practical quantum oblivious transfer. Dans *Advances in Cryptology: Proceedings of CRYPTO '91*, volume 576 de *Lecture Notes in Computer Science*, pages 351–366. Springer Verlag, 1992.

[6] C. H. Bennett, G. Brassard, et J.-M. Robert. Privacy amplification by public discussion. *SIAM Journal on Computing*, 17:210–229, 1988.

[7] E. R. Berlekamp. *Algebraic coding theory.* McGraw-Hill, New York, 1968.

[8] G. Brassard. *Cryptologie contemporaine.* Masson, 1992.

[9] G. Brassard et C. Crépeau. Quantum bit commitment and coin tossing protocols. Dans *Advances in Cryptology: Proceedings of Crypto '90*, volume 537 de *Lectures Notes in Computer Science*, pages 49–61. Springer-Verlag, 1991.

[10] G. Brassard, C. Crépeau, R. Josza, et D. Langlois. A quantum bit commitment scheme provably unbreakable by both parties. Dans *Proceedings of the 34th annual symposium on Foundations of Computer Science*, pages 362–371, Novembre 1993.

[11] G. Brassard et L. Salvail. Secret-key reconciliation by public discussion. Dans *Proceedings of Eurocrypt '93*, volume 765 de *Lecture Notes in Computer Science*, pages 410–423. Springer-Verlag, 1994.

[12] J. Breguet, A. Muller, et N. Gisin. Quantum cryptography with polarized photons in optical fibres. experiment and practical limits. *Journal of Modern Optics*, 41(12):2405–2412, Décembre 1994.

[13] J. L. Carter et M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[14] W. Diffie et M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[15] A. Einstein, B. Podolsky, et N. Rosen. Can quantum-mechanical description of physical reality be considered complete ? *Physical Review*, 47(777–780), 1935.

[16] G. D. Forney, Jr. *Concatenated Codes.* M. I. T. Press, Cambridge, MA, 1966.

[17] M. J. E. Golay. Notes on digital coding. Dans *Proc. IEEE*, volume 37, page 657, 1949.

[18] R. W. Hamming. Error detecting and error correcting codes. *Bell Syst. Tech. J.*, 29:147–160, 1950.

[19] L. Lamport. *LaTeX: A Document preparation system.* Addison-Wesley, 1986.

[20] F. J. MacWilliams et N. J. A. Sloane. *The theory of error-correcting codes.* North-Holland, 1977.

[21] M. O. Rabin. How to exchange secrets by oblivious transfer. Technical Memo TR-81, Aiken Computation Laboratory, Harvard University, 1981.

[22] I. S. Reed et G. Solomon. Polynomial codes over certain finite fields. *J. SIAM*, 8:300–304, 1960.

[23] R. Resnick et D. Halliday. *Ondes, Optique et Physique Moderne*. Éditions du Renouveau Pédagogique, 1980.

[24] P. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. Dans *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Novembre 1994.

[25] W. R. Stevens. *UNIX Networks Programming*. Prentice-Hall Software Series. Prentice-Hall, 1990.

[26] B. Stroustrup. *The C++ programming language*. Addison-Wesley, Juin 1993.

[27] P. D. Townsend et C. Marand. Quantum key distribution over distances as long 30 km. *Optics Letters*, 20, 15 Août 1995.

[28] M. N. Wegman et J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

[29] S. Wiesner. Conjugate coding. *Sigact News*, 15(1):78–88, 1983. manuscrit écrit *circa* 1970.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-LaTeX

Document Log:

Manuscript Version 0

Typeset by $\mathcal{AMS}$-LaTeX — 1 December 1995

Simulation de Protocoles de Cryptographie Quantique

Jean-François Riendeau

Laboratoire d'informatique théorique et quantique
Université de Montréal
CP 6128, succursale CENTRE-VILLE
Montréal (Québec) Canada H3C 3J7
Tel. : (514) 343-6111 poste 3514

E-mail address: riendeauiro.umontreal.ca