

# Projet de Protocoles Internet

Juliusz Chroboczek

18 novembre 2023

## 1. Introduction

Le but de ce projet est de développer un protocole de système de fichiers distribué : chaque pair exporte une arborescence de fichiers (qui peut changer à tout moment), et l'exporte à tous les autres pairs. Le protocole est un protocole hybride :

- un serveur REST central sert de point de rendez-vous;
- le transfert des données se fait par un protocole pair-à-pair basé sur UDP.

Chaque pair est identifié par un nom, qui est une chaîne de caractères arbitraire (par exemple un nom de domaine ou le nom d'un être humain). On suppose les noms de pairs uniques<sup>1</sup>.

Il y a trois endroits où le protocole utilise des techniques cryptographiques :

1. la communication avec le serveur central se fait en HTTP protégé par TLS;
2. les données stockées dans le système de fichiers sont stockées sous forme d'un arbre de Merkle utilisant des *hashes* SHA-256;
3. les messages échangés par les pairs sont optionnellement protégés par des signatures cryptographiques.

Les deux premiers points sont faciles à implémenter, et sont donc obligatoires. Le dernier point demande quelques connaissances cryptographiques de base, et est donc optionnel (paragraphe 6). Le projet est conçu pour être facile à faire en Go, Java et Python. Il est sans-doute faisable en d'autres langages.

## 2. Structures de données

Chaque pair exporte une arborescence de fichiers structurée comme un *arbre de Merkle*<sup>2</sup> : chaque nœud est identifié par un *hash* cryptographique qui couvre tout ce qui est contenu ainsi que les *hashes* de tous ses fils. Le *hash* de la racine change donc à chaque fois qu'un nœud de l'arbre change.

Il existe trois types de nœuds dans l'arborescence :

---

1. Le sous-protocole cryptographique optionnel décrit à la partie 6 garantit que les noms de pairs sont uniques.  
2. [https://en.wikipedia.org/wiki/Merkle\\_tree](https://en.wikipedia.org/wiki/Merkle_tree)

- les nœuds *chunk*, qui contiennent simplement des suites d’octets; les fichiers de moins d’un kilooctet sont représentés par des nœuds *chunk*;
- les nœuds *big file*, qui ont de 2 à 32 fils, dont chacun est lui-même soit un *big file* soit un *chunk*, représentent les fichiers ayant une taille de 1024 octets ou plus;
- les nœuds *directory*, qui ont de 0 à 16 fils, représentent les répertoires.

Pour simplifier l’implémentation, le protocole ne supporte pas les répertoires contenant plus de 16 fichiers ou répertoires. La représentation des nœuds sous forme de suites d’octets est donnée au paragraphe 5.2.

### 3. Description informelle du protocole

Le protocole consiste de deux parties : un protocole REST client-serveur, basé sur HTTPS, et un protocole pair-à-pair basé sur UDP. Le serveur participe aussi au protocole pair-à-pair.

**Découverte de pairs** La découverte des pairs se fait à travers l’API REST, qui permet d’obtenir la liste de tous les pairs du réseau, et, pour chaque pair, ses adresses et (optionnellement) sa clé cryptographique.

**Enregistrement auprès du serveur** Pour que les autres pairs puissent le découvrir, un pair doit s’enregistrer auprès du serveur. Pour cela, il découvre la ou les adresses du serveur puis, depuis chacune de ses adresses IP, il envoie un message *Hello* à l’une des adresses du serveur. Le serveur répond par un message *HelloReply*, ce qui permet d’implémenter l’enregistrement de manière fiable (le pair peut réémettre *Hello* tant qu’il ne reçoit pas de *HelloReply*).

Après avoir reçu le *Hello* du pair, le serveur va confirmer que le pair reçoit bien les messages du serveur. Pour cela, il envoie un message *PublicKey* au pair, qui répond par un *PublicKeyReply*. Le serveur n’annonce l’adresse IP du pair sur l’interface REST que lorsqu’il aura reçu un *PublicKeyReply* avec la bonne *Id*.

Le serveur enverra ensuite au pair un message *Root* pour obtenir la valeur de sa racine. Si le pair n’exporte pas d’arborescence, il répond avec le *hash* de la chaîne vide; s’il exporte une arborescence, il répond avec le *hash* de sa racine.

Pour maintenir l’enregistrement, le pair doit continuer à envoyer des messages *Hello* ou *Root* au serveur toutes les 30 secondes environ. Si le serveur n’a reçu aucun message du pair pendant 180 secondes, il supprime toutes les données à propos du pair.

**Transfert de données** Pour télécharger l’arborescence d’un pair *q*, un pair *p* commence par découvrir les adresses de *q* ainsi que la valeur de son *hash* racine, soit en consultant l’API REST, soit en demandant directement à *q*<sup>3</sup>. Une fois le *hash* racine obtenu, il envoie des messages *GetDatum* à *q*; chacun de ces messages permet de télécharger un des nœuds de l’arborescence de *q*. Le pair *p* doit *obligatoirement* vérifier les *hashes* des données qu’il reçoit, ce qui empêche un attaquant

---

3. Si le pair n’implémente pas les signatures cryptographiques, il est préférable d’utiliser l’API REST, qui est protégée par TLS.

d'injecter des données incorrectes. De proche en proche, le pair  $p$  obtient la partie de l'arborescence qui l'intéresse.

Il se peut que l'arborescence de  $q$  change pendant que  $p$  la télécharge ;  $p$  s'en rend compte lorsqu'il reçoit un message *NoDatum* indiquant qu'une donnée n'existe plus. Dans ce cas, il peut soit abandonner (et afficher un message d'erreur à l'utilisateur), soit demander au serveur ou au pair la nouvelle valeur de la racine et recommencer.

Comme une donnée est identifiée par son *hash*, le protocole garantit qu'elle ne change jamais sans changer d'identifiant. Le récepteur peut donc cacher indéfiniment (tant qu'il a assez de mémoire) les données qu'il reçoit.

**Protection cryptographique** Les *hashes* contenus dans l'arbre de Merkle protègent le contenu des nœuds : un attaquant ne peut pas injecter une donnée incorrecte, car cela invaliderait les *hashes* de l'arbre. Le point faible est le *hash* racine : si un attaquant arrive à injecter un *hash* racine incorrect, il peut injecter des données arbitraires.

Le transfert des *hashes* racine dans le sens serveur vers pair se fait à travers l'API REST, qui est protégée par TLS. Ce n'est pas le cas du transfert dans le sens pair vers serveur, ou du transfert direct entre pairs, qui, lui, se fait en UDP. Pour éviter ce problème, le protocole permet optionnellement de protéger les transferts par une signature cryptographique. Ce sous-protocole est décrit au paragraphe 6.

**Traversée de NAT** Le protocole pair-à-pair utilise un échange direct de paquets UDP. Pour permettre aux pairs qui se trouvent derrière un NAT ou un *firewall* de communiquer, le protocole inclut un mécanisme de traversée de NAT à travers le serveur.

Après qu'un pair  $p$  a envoyé plusieurs messages *Hello* au pair  $q$  et n'a pas reçu de réponse, il initie une procédure de traversée de NAT. Il commence par envoyer un message *NatTraversalRequest* au serveur qui contient l'adresse de  $q$ . Le serveur envoie alors un message *NatTraversal* au pair  $q$  qui contient l'adresse de  $p$  (l'adresse source du premier message). Lorsqu'il reçoit ce message, le pair  $q$  envoie à  $q$  un message *Hello*. Si  $p$  reçoit ce *Hello*, il envoie un *HelloReply* comme d'habitude, puis envoie un *Hello* pour déterminer si la traversée de NAT a réussi.

La traversée de NAT utilise un protocole non-fiable (les messages ne sont pas acquittés), il est donc nécessaire de rémettre la requête plusieurs fois. Cependant, la procédure peut prendre du temps : le pair  $q$  peut avoir besoin de contacter le serveur pour obtenir la clé cryptographique de  $p$  ; il est donc nécessaire d'utiliser des *timeouts* suffisamment grands.

## 4. Description du protocole client-serveur

Le serveur implémente un protocole de type REST qui sert notamment à localiser les autres pairs. Vous devrez implémenter le côté client de ce protocole, mais pas le côté serveur.

### 4.1. Liste de pairs

Pour obtenir la liste des noms des pairs connus du serveur, un client fait une requête GET à l'URL `/peers/`. Le serveur répond 200 avec le corps contenant une liste de noms de pairs, un

par ligne.

## 4.2. Adresses de pairs

Pour localiser un pair  $p$ , un client fait une requête GET à l'URL `/peers/ $p$ /addresses`. Le serveur répond :

- 200 si le pair est connu, et alors le corps contient une liste d'adresses de socket UDP, une par ligne;
- 404 si le pair  $p$  n'est pas connu.

## 4.3. Recherche de clés

Pour trouver la clé publique d'un pair  $p$ , un client fait une requête GET à l'URL `/peers/ $p$ /key`. Le serveur répond :

- 200 si le pair est connu et a annoncé une clé publique, et alors le corps contient la clé (une suite de 64 octets, voyez le paragraphe 6);
- 204 si le pair est connu, mais n'a pas annoncé de clé publique;
- 404 si le pair n'est pas connu.

## 4.4. Recherche de racines

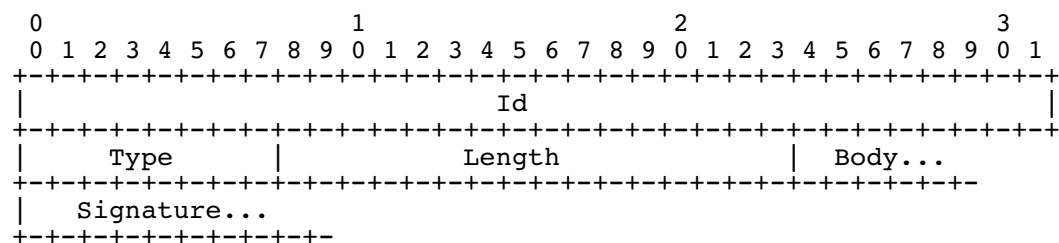
Pour demander au serveur le dernier *hash* connu de la racine de l'arborescence du pair  $p$ , un client fait une requête à l'URL `/peers/ $p$ /root`. Le serveur répond :

- 200 si le pair est connu et a annoncé une racine, et alors le corps contient le *hash* racine (une suite de 32 octets);
- 204 si le pair est connu, mais n'a pas annoncé de racine au serveur;
- 404 si le pair n'est pas connu.

## 5. Protocole pair-à-pair

Le serveur et tous les pairs implémentent un protocole pair-à-pair basé sur UDP. Le protocole est symétrique : étant donnés deux pairs  $p$  et  $q$ , chacun de  $p$  et de  $q$  peut envoyer une requête à tout moment, ce qui entraîne qu'un pair peut recevoir une requête juste après en avoir envoyé une. Le protocole n'est pas strictement un protocole requête réponse : certaines requêtes sollicitent une réponse, d'autres n'en sollicitent pas (ce sont de pures indications).

Tous les messages ont le format suivant :



Le champ *Type* indique le type du message; les valeurs 0 à 127 indiquent des requêtes, les valeurs 128 à 255 indiquent des réponses. Le champ *Id* a le comportement suivant :

- dans une requête, il a une valeur arbitraire (choisie par l'émetteur);
- dans une réponse, il a la même valeur que dans la requête à laquelle le message répond.

Le champ *Length* indique la longueur en octets du champ *Body*.

Le corps peut optionnellement être suivi d'une signature cryptographique (paragraphe 6). Un pair qui n'implémente pas les signatures cryptographiques n'inclut pas de signature dans les messages qu'il envoie, et ignore les signatures dans les messages qu'il reçoit.

## 5.1. Format des messages

### 5.1.1. Pas d'opération

Le message de type *NoOp* = 0 n'a aucun effet, il est ignoré par le récepteur. Le champ *Type* vaut 0, le corps a une taille arbitraire entre 0 et 1024 octets.

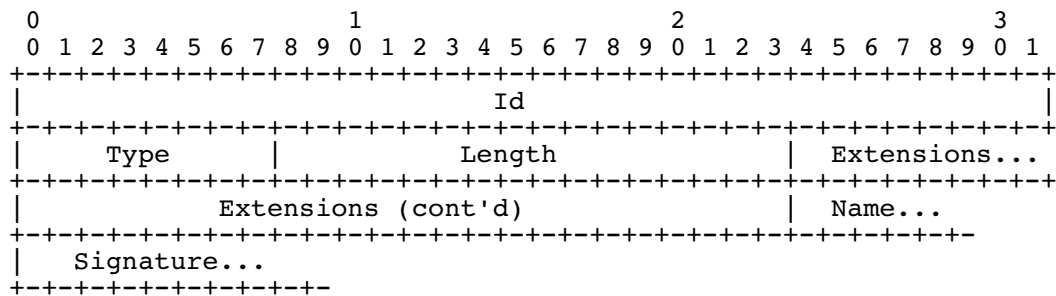
### 5.1.2. Erreurs

Le message de type *Error* = 1 sert à envoyer un message d'erreur. Le corps est un message lisible par un être humain, codé en UTF-8. Le message de type *ErrorReply* = 128 sert à envoyer un message d'erreur en réponse à une requête. Il a la même syntaxe que le message *Error*.

Mon pair envoie des messages *Error* et *ErrorReply* lorsqu'il détecte des *bugs* de votre implémentation, et je vous encourage fortement à en éficher le contenu.

### 5.1.3. Établissement d'association

Les messages de type *Hello* = 2 et *HelloReply* = 129 servent à maintenir les associations entre pairs. Ils ont la structure suivante :



Le champ *Extensions*, de longueur 4, indique les extensions supportées par ce pair (voir paragraphe 7); si votre pair ne supporte aucune extension, il enverra ce champ égal à 0 et l'ignorera lors de la réception. Le champ *Name* indique le nom du pair qui émet le message.

Il est *obligatoire* de répondre à tout message *Hello* par un message *HelloReply* ayant le même *Id*. Il est *obligatoire* d'envoyer un message *HelloReply* à un pair avant de lui envoyer tout autre message : un pair peut ignorer les messages reçus de la part d'un pair qui n'a pas correctement

effectué un *handshake* initial<sup>4</sup>. Un pair a le droit d'oublier l'association avec un autre pair après qu'il n'a reçu aucun message de ce dernier depuis 180 s : il est donc nécessaire de répéter l'échange initial après un temps de silence.

#### 5.1.4. Transfert de clés

Les messages *PublicKey* = 3 et *PublicKeyReply* = 130 servent à échanger les clés publiques. Si l'émetteur n'implémente pas les signatures cryptographiques, ces messages ont un corps vide (*Length* = 0). Si l'émetteur implémente les signatures cryptographiques, leur corps contient la clé publique de l'émetteur (*Length* = 64).

Il est *obligatoire* de répondre à tout message *PublicKey* par un message *PublicKeyReply* ayant le même *Id*, même si l'on n'implémente pas les signatures cryptographiques.

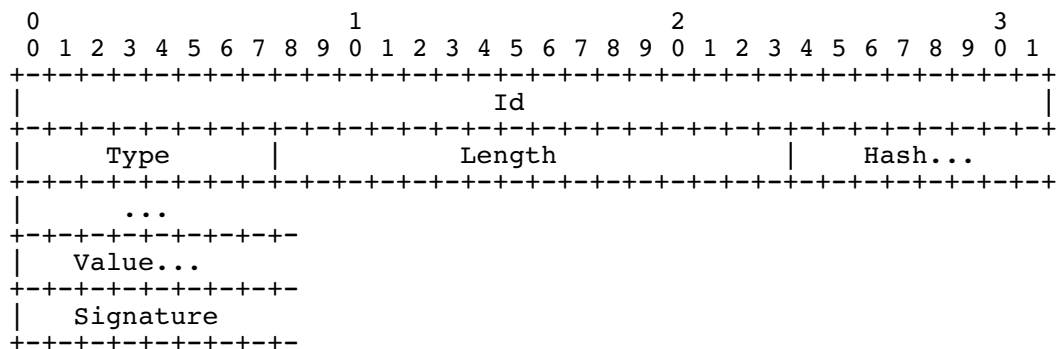
#### 5.1.5. Transfert de racines

Les valeurs des *hashes* racine sont communiquées à l'aide des messages *Root* = 4 et *RootReply* = 131. Le corps de ces messages contient le *hash* racine de l'émetteur. Si l'émetteur n'exporte pas de données, il annonce le *hash* de la chaîne vide.

#### 5.1.6. Transfert de données

Chaque pair maintient des données, dont chacune est identifiée par son *hash* SHA-256. Un pair peut demander à un pair la valeur d'une donnée en envoyant un message de type *GetDatum* = 5 dont le corps contient un *hash* (*Length* = 32).

Si le pair a la donnée demandée, il répond par un message *Datum* = 132 ayant le format suivant :



Le champ *Hash* contient le *hash* du champ *Value*; il a une longueur de 32 octets. Le champ *Value* contient la valeur elle-même, et a une longueur de *Length* - 32 octets.

Si le pair n'a pas la donnée, il répond par un message *NoDatum* = 133 contenant le *hash* demandé (*Length* = 32).

Il est *obligatoire* de répondre à tous les messages *GetDatum*, même si l'on n'implémente pas le transfert de données ou si la donnée demandée n'existe pas. Il est *obligatoire* de vérifier la valeur du *hash* avant de se servir des données contenues dans un message *Datum*.

4. Par exemple pour éviter les attaques par amplification.

### 5.1.7. Traversée de NAT

Les messages *NatTraversalRequest* = 6 et *NatTraversal* = 7 servent à la traversée de NAT (paragraphe 3). Le corps de ces requêtes contient des adresses de *socket* UDP (une adresse IP suivie immédiatement d'un numéro de port). Le champ *Length* vaut 6 pour une adresse IPv4, et 18 pour une adresse IPv6.

## 5.2. Format des données

Du point de vue du protocole, les données transférées dans les messages *Datum* n'ont pas de structure, ce sont simplement des suites d'octets. Cependant, les données que nous transférons ont un format bien défini. Le premier octet de la donnée indique le type de la donnée :

- *Chunk* = 0 indique un bloc d'au plus 1024 octets de données. L'octet de type est immédiatement suivi des données. Les fichiers de taille inférieure (ou égale) à 1024 sont codés par une seule donnée de type *Chunk*.
- *Tree* = 1 indique que la donnée représente un nœud interne d'un arbre dont les feuilles sont des *Chunk*. L'octet de type est suivi d'une suite d'au plus 32 *hashes* de 32 octets chacun ;
- *Directory* = 2 indique que la donnée représente un répertoire. L'octet de type est immédiatement suivi d'une suite d'au plus 16 entrées de répertoire de longueur 64 dont chacune a la structure suivante :
  - *Name*, 32 octets, indique le nom du fichier, complété par des octets de valeur 0 ;
  - *Hash*, 32 octets, indique le *hash* du fichier référencé par cette entrée ;
- les types 3 à 255 sont réservés pour des extensions futures ; si votre pair reçoit une donnée de type inconnu, il devra afficher un message d'erreur.

Pour définir un nouveau type de donnée, vous devrez envoyer un message expliquant l'extension à la liste du projet, et je vous affecterai un type de donnée.

## 6. Signatures cryptographiques

Le protocole peut optionnellement protéger les messages qu'il envoie par des signatures générées selon l'algorithme ECDSA sur la courbe elliptique P-256 avec la fonction de hachage SHA-256. Il y a trois concepts à connaître :

- une clé privée permet de générer des signatures ; les clés privées n'apparaissent pas dans le protocole (elles sont privées à un pair) ;
- une clé publique permet de vérifier les signatures ; les clés publiques sont représentées comme des chaînes de 64 octets (32 octets pour chacune des composantes *x* et *y*) ;
- une signature est une chaîne de 64 octets.

Les signatures sont calculées sur le message entier mais sans la signature, c'est-à-dire sur les octets 0 à 6 + *Length*. Pour simplifier le protocole, les signatures ne prennent en compte ni un *pseudo-header* ni un *nonce*, ce qui rend le protocole vulnérable aux attaques par rejeu.

Comme l'intégrité des données transférées est garantie par les arbres de Merkle, il n'est pas nécessaire de signer tous les messages. Pour un pair qui implémente les signatures cryptographiques, il est *obligatoire* de signer les messages suivants :

- les messages *Hello* et *HelloReply* (la signature empêche un autre pair de se faire passer pour un pair existant);
- les messages *PublicKey* et *PublicKeyReply* (la signature constitue une preuve de la possession de la clé privée<sup>5</sup>);
- les messages *Root* et *RootReply* (la racine est le point d’ancrage des arbres de Merkle).

Un pair qui implémente les signatures cryptographiques et qui reçoit un message d’un pair dont la clé publique est publiée sur le serveur doit *obligatoirement* ignorer ce message s’il a l’un des types ci-dessus et s’il n’est pas signé. Il doit naturellement ignorer tout message signé incorrectement. Des conseils d’implémentation sont donnés à l’annexe A.

## 7. Mécanisme d’extension

Le protocole est extensible. Chaque extension au protocole a un numéro  $n$ , compris entre 0 et 31, et définit les messages numéro  $64 + n$  et  $192 + n$ .

Le champ *Extensions* des messages *Hello* et *HelloReply* indique l’ensemble des extensions implémentées par le pair qui l’émet sous forme d’un entier de 32 bits : si le bit de valeur  $2^n$  vaut 1, alors l’émetteur du message implémente l’extension numéro  $n$ .

Pour définir une extension, vous devez envoyer un mail à la liste du projet, et je vous affecterai un numéro d’extension.

## 8. Sujet minimal

Au minimum, votre pair devra être capable de :

- découvrir les adresses des pairs enregistrés auprès du serveur;
- étant donné un nom de pair  $p$  et un *hash*  $h$ , télécharger la donnée identifiée par  $h$  et stockée dans le pair  $p$ ;
- recevoir un fichier distant identifié par un chemin.

Il est *obligatoire* de vérifier les *hashes* des données reçues. Il est *obligatoire* de respecter le protocole décrit dans ce document : les programmes qui n’interopèrent pas avec mon pair ne seront pas acceptés.

Le choix de l’interface utilisateur est laissé à votre discrétion : vous pourrez par exemple implémenter un programme en ligne de commande, une interface interactive au clavier, une interface graphique, ou une interface *web*. Ne passez pas trop de temps sur l’interface utilisateur, c’est un projet de réseaux, pas un projet d’interfaces graphiques.

Points en plus si votre pair se comporte bien en présence de pertes de paquets ou de problèmes réseau (en rémettant les paquets, en essayant toutes les adresses du pair distant et en retenant lesquelles ont fonctionné récemment). Points en plus si votre pair participe au protocole de traversée de NAT. Points en plus si votre pair est capable d’exporter une arborescence de fichiers (de permettre aux autres pairs de la télécharger). Points en plus si votre pair implémente les signatures cryptographiques. Points en plus s’il implémente des extensions au sujet tout en restant compatible avec un pair non-étendu.

---

5. En fait, ce n’est pas entièrement vrai, car le protocole est vulnérable au rejeu.



## 9. Extensions

Quelques idées d'extensions.

**Mises à jour et notifications asynchrones** Une solution simple au projet ne permet pas de mettre à jour les fichiers : au lancement, votre pair exporte une arborescence qui ne peut pas changer. Une solution plus intéressante permet de modifier un fichier à l'exécution, ou d'ajouter un fichier. Dans ce cas, il faudra recalculer tous les *hashes* de la branche qui va du fichier modifié jusqu'à la racine.

Lors d'une telle modification, il faut notifier le serveur du changement de la valeur de la racine. Vous remarquerez que le message *RootReply* constitue un acquittement du message *Root* correspondant, ce qui permet de transférer la racine de manière fiable.

**Caches** Une donnée est identifiée par son *hash*, qui change forcément si la donnée change. Il est donc facile d'implémenter un cache qui évite de contacter un pair distant pour chaque requête de donnée. Si vous implémentez un cache de données, assurez-vous qu'il ne croît pas sans borne.

**Pipelining et contrôle de congestion** Un pair simple n'a jamais plus d'une requête en vol, ce qui rend le transfert de fichiers assez lent. Cependant, si votre pair est capable de maintenir plusieurs requêtes en vol, il faudra qu'il implémente un algorithme de contrôle de congestion.

Je vous propose l'algorithme suivant, qui est classique est simple à implémenter. Pour chaque pair distant, votre pair maintient une variable entière, la *taille de la fenêtre*, qui est le nombre maximal de requêtes en vol autorisées. À chaque fois qu'une réponse est reçue, la fenêtre augmente de 1 ; à chaque fois qu'une requête est réemise, la fenêtre est divisée par 2 (sans descendre en dessous de 1).

**Répertoires de plus de 16 entrées** Le protocole limite la taille des répertoires à 16 entrées. Il serait utile de définir une extension qui permet de transférer des répertoires plus gros (il faudra définir un nouveau type de donnée).

**Clés de session** Le protocole cryptographique décrit au paragraphe 6 effectue une opération sur les courbes elliptiques pour chaque message signé. Il serait plus efficace d'utiliser un algorithme de signature symétrique (par exemple HMAC) avec une clé de session.

Pour cela, on pourra par exemple définir une extension au protocole qui effectue un échange Diffie-Hellman authentifié ; la clé ainsi négociée pourra servir de clé de session spécifique à une paire de pairs.

**Chiffrage** Le protocole envoie toutes les données en clair. Il serait souhaitable de les chiffrer afin de garantir la confidentialité des échanges. Ce n'est a priori pas très facile. Il faudra tout d'abord implémenter un algorithme d'échange de clés, comme ci-dessus ; et il faudra ensuite définir un algorithme de chiffrement pour tous les messages.

**Autres extensions** Toutes les extensions seront les bienvenues à condition qu'elles restent compatibles avec le protocole de base.

## 10. Modalités de rendu

Le projet sera fait par groupes de deux étudiants au plus. Vous me soumettez une archive `.tar.gz` contenant :

- un programme implémentant la partie du sujet que vous aurez traitée;
- un rapport de quelques pages au format PDF qui m'indique les choix d'implémentation que vous aurez faits, les algorithmes que vous aurez utilisés, les extensions au sujet que vous voudriez que je remarque, et toute autre information qui peut m'intéresser.

L'archive devra s'appeler `nom1-nom2.tar.gz`, et s'extraire dans un sous-répertoire `nom1-nom2` du répertoire courant. Par exemple, si vous vous appelez *Arlo Guthrie* et *Janis Joplin*, votre archive devra porter le nom `guthrie-joplin.tar.gz` et son extraction devra créer un répertoire `guthrie-joplin` contenant tous les fichiers que vous me soumettez.

### A. Annexe : implémentation des primitives cryptographiques

Une clé publique ECDSA est une paire d'entiers  $(x, y)$ . Une signature est une paire d'entiers  $(r, s)$ . Dans ce projet, nous représentons ces paires d'entiers par des chaînes de 64 octets, où les premiers 32 représentent le premier entier et les derniers 32 le deuxième.

Dans cette partie, je donne une implémentation des constructions nécessaires à l'implémentation de l'extension décrite au paragraphe 6 en Python, Go et Java.

#### A.1. Python

Pour générer une clé privée :

```
import ecdsa
import hashlib

privateKey = ecdsa.SigningKey.generate(
    curve=ecdsa.SECP256k1, hashfunc=hashlib.sha256,
)
```

Pour obtenir la clé publique associée :

```
publicKey = privateKey.get_verifying_key()
```

Pour formater la clé publique comme une chaîne de 64 octets :

```
publicKey.to_string()
```

Pour *parser* une clé publique représentée comme une chaîne de 64 octets :

```

publicKey = ecdsa.VerifyingKey.from_string(
    body, curve=ecdsa.SECP256k1, hashfunc=hashlib.sha256,
)

```

Pour calculer la signature d'un message :

```
signature = privateKey.sign(data)
```

Pour vérifier un message :

```
ok = publicKey.verify(signature, data)
```

## A.2. Go

Pour générer une clé privée :

```

import (
    "crypto/ecdsa"
    "crypto/elliptic"
    "crypto/rand"
)

privateKey, err :=
    ecdsa.GenerateKey(elliptic.P256(), rand.Reader)

```

Pour obtenir la clé publique associée :

```
publicKey, ok := privateKey.Public().(*ecdsa.PublicKey)
```

Pour formater la clé publique comme une chaîne de 64 octets :

```

formatted := make([]byte, 64)
publicKey.X.FillBytes(formatted[:32])
publicKey.Y.FillBytes(formatted[32:])

```

Pour *parser* une clé publique représentée comme une chaîne de 64 octets :

```

import "math/big"

var x, y big.Int
x.SetBytes(data[:32])
y.SetBytes(data[32:])
publicKey := ecdsa.PublicKey{
    Curve: elliptic.P256(),
    X: &x,
    Y: &y,
}

```

Pour calculer la signature d'un message :

```

import (
    "crypto/sha256"
    "crypto/rand"
)

hashed := sha256.Sum256(data)
r, s, err := ecdsa.Sign(rand.Reader, privateKey, hashed[:])
signature := make([]byte, 64)
r.FillBytes(signature[:32])
s.FillBytes(signature[32:])

```

Pour vérifier un message :

```

var r, s big.Int
r.SetBytes(signature[:32])
s.SetBytes(signature[32:])
hashed := sha256.Sum256(data)
ok = ecdsa.Verify(publicKey, hashed[:], &r, &s)

```

### A.3. Java

Cette partie n'a pas été testée.

```

import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.security.Signature;

```

Pour générer une clé privée :

```

ECGenParameterSpec ecSpec = new ECGenParameterSpec("secp256k1");
KeyPairGenerator g = KeyPairGenerator.getInstance("EC");
g.initialize(ecSpec, new SecureRandom());
KeyPair keypair = g.generateKeyPair();
PrivateKey privateKey = keypair.getPrivate();

```

Pour obtenir la clé publique associée :

```

PublicKey publicKey = keypair.getPublic();

```

Pour formater la clé publique comme une chaîne de 64 octets :

```

BigInteger x = publicKey.getW().getAffineX();
BigInteger y = publicKey.getW().getAffineY();
byte[] xbytes = x.toByteArray();
byte[] ybytes = y.toByteArray();
byte[] publicBytes = new byte[64];
System.arraycopy(xbytes, 0, publicBytes,
                 32 - xbytes.length, xbytes.length);
System.arraycopy(ybytes, 0, publicBytes,
                 64 - ybytes.length, ybytes.length);

```

Pour *parser* une clé publique représentée comme une chaîne de 64 octets :

```

KeyFactory kf = KeyFactory.getInstance("EC");
byte[] xbytes = Arrays.copyOfRange(publicBytes, 0, 32);
byte[] ybytes = Arrays.copyOfRange(publicBytes, 32, 64);
BigInteger x = BigInteger(xbytes);
BigInteger y = BigInteger(ybytes);
ECPublicKeySpec keySpec =
    new ECPublicKeySpec(new ECPoint(x, y), ecSpec);
publicKey = kf.generatePublic(keySpec);

```

Pour calculer la signature d'un message :

```

Signature ecdsaSign =
    Signature.getInstance("SHA256withECDSA");
ecdsaSign.initSign(privateKey);
ecdsaSign.update(data);
byte[] signature = ecdsaSign.sign();

```

Pour vérifier un message :

```

Signature ecdsaVerify =
    Signature.getInstance("SHA256withECDSA");
KeyFactory kf = KeyFactory.getInstance("EC");
ecdsaVerify.initVerify(publicKey);
ecdsaVerify.update(message);
boolean result =
    ecdsaVerify.verify(Base64.getDecoder().decode(signature));

```

#### A.4. Autres langages

Les constructions ci-dessus sont sans doute possibles à effectuer dans n'importe quel langage qui dispose d'une bonne bibliothèque ECDSA. La seule difficulté consiste à formater les clés publiques et les signatures dans le bon format : il faut représenter les paires d'entiers par des chaînes de 64 octets, comme décrit ci-dessus.