Projet de Protocoles Internet

Juliusz Chroboczek

27 novembre 2022

1. Introduction

Le but de ce projet est d'implémenter un service de *micro-blogging* (comme par exemple *Twitter*) mais implémenté de manière hybride :

- un serveur (HTTPS et UDP) sert à la localisation des pairs et à la distribution des clés cryptographiques ;
- le transfert de messages est effectué directement entre les pairs.

Le protocole inclut des provisions pour la protection cryptographique des données, mais ces provisions sont optionnelles afin de permettre aux étudiants qui le désirent d'avoir une note correcte sans s'intéresser à la cryptographie. Cela rend le protocole vulnérable aux *downgrade attacks*.

Le sujet ne définit que le protocole, et vous êtes libres d'implémenter l'interface utilisateur qui vous convient. Par exemple, un groupe pourra implémenter un simple programme qui affiche les messages des autres utilisateurs au fur et à mesure qu'ils arrivent, tandis qu'un autre groupe pourra implémenter une application interactive qui permet de visualiser et de publier des messages à la demande.

2. Description du protocole

Cette partie décrit le comportement du protocole. Les détails du format des messages échangés sont donnés dans la partie 3.

2.1. Structure de données

Chaque pair publie une suite de messages. La suite de messages est codée sous forme d'un arbre (d'arité de plus 32), dont la frontière est la suite de messages ; en d'autres termes, pour obtenir la suite de messages publiée par un pair, il faut faire un parcours en profondeur de l'arbre et énumérer la suite des feuilles.

Naturellement, il existe plusieurs arbres qui codent la même suite de messages. Un pair peut publier un arbre dégénéré (une liste chaînée) ou un arbre plus complexe. Par contre, chaque pair doit être capable d'interpréter un arbre arbitraire.

Afin de transmettre les données incrémentalement à travers le réseau, ces dernières sont codées par un *arbre de Merkle* : chaque sommet est représenté par un *hash* de sa représentation, et une arête est représentée par le *hash* qui identifie le fils.

2.2. Enregistrement auprès du serveur

Initialement, un pair fait une requête HTTPS au serveur pour lui communiquer :

- son nom d'utilisateur ;
- (optionnellement) la clé publique du pair.

Le pair effectue ensuite un ou plusieurs échanges UDP avec le serveur, ce qui établit la ou les adresses de socket UDP du pair et prouve (optionnellement) la possession de la clé privée correspondant à la clé publique publiée.

Le serveur oublie un pair au bout d'une heure. Un pair qui désire rester enregistré auprès du serveur doit donc répéter chaque échange UDP toutes les 55 minutes au plus.

2.3. Établissement d'une session pair-à-pair

Une session est une relation asymétrique entre un pair A et un pair B : si un pair A a établi une session avec un pair B, A peut envoyer des requêtes et des messages non-solicités au pair B, et B peut envoyer des réponses à A. Il est bien sûr possible d'établir deux sessions, une entre A et B, et une entre B et A.

Pour établir une session, un pair A envoie un message *Hello* au pair B qu'il désire contacter. Le pair B répond par un message *HelloReply*, et la session est établie. (Il se peut, naturellement, que le *Hello* ou le *HelloReply* soit perdu, il faut donc que A réémette sa requête après un *timeout*.) Une session expire au bout d'une heure. Un pair qui désire maintenir une session doit donc répéter l'échange au bout de 55 minutes au plus.

Si A signe ses messages, il se peut que B ne connaisse pas encore la clé publique de A lorsqu'il reçoit un *Hello* ; il doit dans ce cas faire une requête HTTP au serveur pour l'obtenir. Le *timeout* initial du *Hello* doit donc être suffisamment grand pour laisser le temps au pair B d'apprendre la clé publique (par exemple s).

2.4. Transfert de données

Lorsqu'il désire télécharger la liste des messages d'un pair B, le pair A établit une session avec B puis transfère l'arbre des données depuis la racine qui lui a été communiquée par le serveur. Si une partie de l'arbre n'a pas changé depuis la dernière fois que A l'a téléchargée, A le détecte en comparant les *hashes* de l'arbre de Merkle, et se sert des données téléchargées auparavant.

2.5. Traversée de NAT

Si A n'arrive pas à contacter directement un pair B, il peut, optionnellement, effectuer une traversée de NAT. Pour cela, A envoie un message non-solicité *NAT Traversal Client* au serveur ; le serveur envoie un message non-solicité *NAT Traversal Server* au pair B, qui réagit en envoyant un message *HelloReply* non-solicité à A. Un peu plus tard, A envoie une requête *Hello* à B ; lorsque B a répondu par *HelloReply*, la session est établie malgré la présence de NAT.

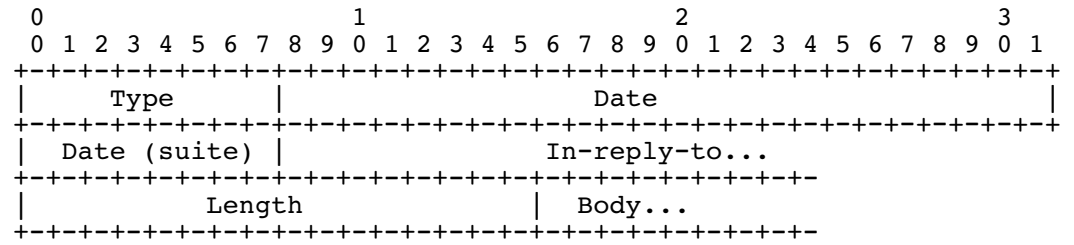
3. Détails du protocole

3.1. Format des données

Les données sont stockées sous forme d'un arbre de Merkle. Cet arbre a deux types de nœuds : les messages et les nœuds internes.

Chaque nœud est stocké comme une suite d'octets. Le *hash* d'un nœud est la suite de 32 octets obtenue en appliquant l'algorithme *SHA-256* à la suite d'octets qui représente le nœud.

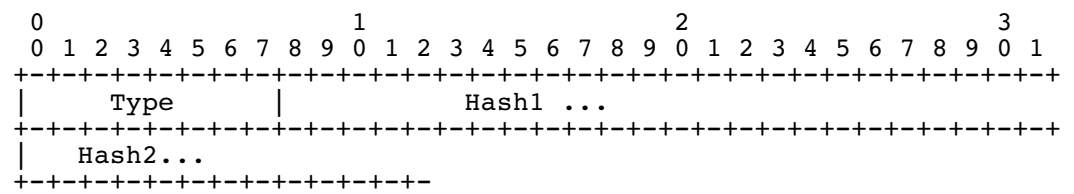
Message Un message est représenté par la structure de données suivante :



- **Type** vaut 0 et indique qu'il s'agit d'un message. Ce champ a une taille d'un octet.
- **Date** indique la date du message, codée comme un nombre de secondes depuis le premier janvier 2022 (l'entier est non-signé, le protocole ne souffre donc pas du problème de l'an 2036) ; ce champ a une taille de 4 octets.
- **In-reply-to** indique le *hash* du message auquel ce messae répond ; il vaut 0 si ce message ne répond pas à un autre message ; ce champ a une taille de 32 octets.
- **Length** indique la taille, en octets, du champ *Body* ; ce champ a une taille de 2 octets.
- **Body** est le message lui-même, codé en UTF-8.

Du fait des limitations d'UDP, les messages sont limités à une taille de 1024 octets.

Internal Un nœud interne est représenté par la structure de données suivante :



- **Type** vaut 1 et indique qu'il s'agit d'un nœud interne. Ce champ a une taille d'un octets.
- Le type est suivi d'une suite de 2 à 32 *hashes* qui identifient les fils du nœud. Chaque *hash* a une taille de 32 octets.

Un pair ne doit jamais publier un nœud interne contenant strictement plus de 32 hashes.

Autres types de nœuds Un nœud peut publier des nœuds dont le champ *Type* ne vaut ni 0 ni 1. Un pair qui reçoit un tel type de nœud doit ignorer le nœud mais ne doit pas arêter son parcours d'arbre.

3.2. Protocole client-serveur

Adresse UDP du serveur Le serveur participe au protocole pair-à-pair. L'adresse du serveur peut être obtenue en faisant une requête `GET` à l'URL `/udp-address`.

Clé publique du serveur La clé publique que le serveur utilise pour signer les messages est disponible à l'URL `/server-key`. Si un GET à cette URL retourne 404, le serveur ne signe pas ses messages.

Enregistrement auprès du serveur Un pair qui désire s'enregistrer auprès du serveur envoie une requête `POST` à l'URL `/register`. Le corps de la requête contient un objet JSON ayant la structure suivante :

```
{
    "name": ...,
    "key": ...
}
```

Le champ *name* contient le nom du pair (par exemple `"elon"` ou `"elon.musk@tesla.com"`). Le champ *key* est optionnel ; si présent, il contient la clé publique du pair (une suite de 64 octets) codée en base 64 (sans *padding*).

Avant d'être effectivement visible, le pair doit établir une session UDP avec le serveur, comme décrit dans le paragraphe 2.3. Si le pair a plusieurs adresses de *socket* (par exemple parce qu'il supporte IPv4 et IPv6 ou parce qu'il a plusieurs connexions à l'Internet), il doit établir une session depuis chacune de ses adresses.

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

Addresses de pairs Pour localiser un pair nommé *p*, un client fait une requête `GET` à l'URL `/peers/`*p*. Le serveur répond avec un code 404 si le pair n'est pas connu, et 200 si le pair est connu. Dans ce dernier cas, le corps de la réponse contient un objet JSON ayant la structure suivante :

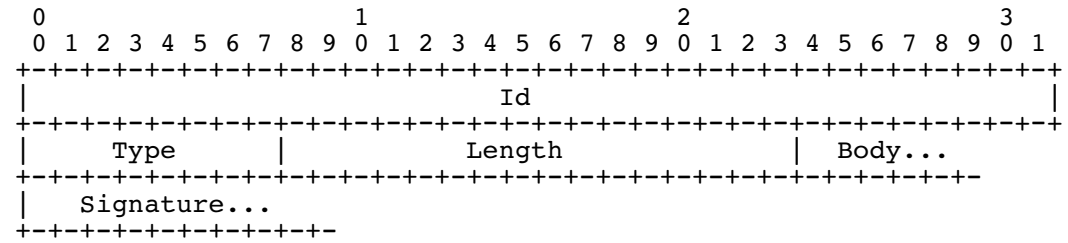
```
{
    "username": ...,
    "addresses": [{
        ip: ...,
        port: ...
    }],
    "key": ...
}
```

Le champs *username* et *key* sont comme ci-dessus. Le champ *adresses* contient la liste des adresses de *socket* du pair, dont chacune est elle même représentée comme un objet JSON.

3.3. Protocole pair-à-pair

Le serveur et tous les pairs implémentent un protocole pair-à-pair basé sur UDP. Comme UDP est non-fiable, il faudra répéter les requêtes lorsqu'on ne reçoit pas de réponse au bout d'un certain temps.

Tous les messages ont le format suivant :



Le champ *Id* contient un ID arbitraire. Dans une requête, il est choisi par l'émetteur (et ne peut pas valoir 0). Dans une réponse, il est recopié depuis la requête. Dans un message non-solicité, il vaut 0.

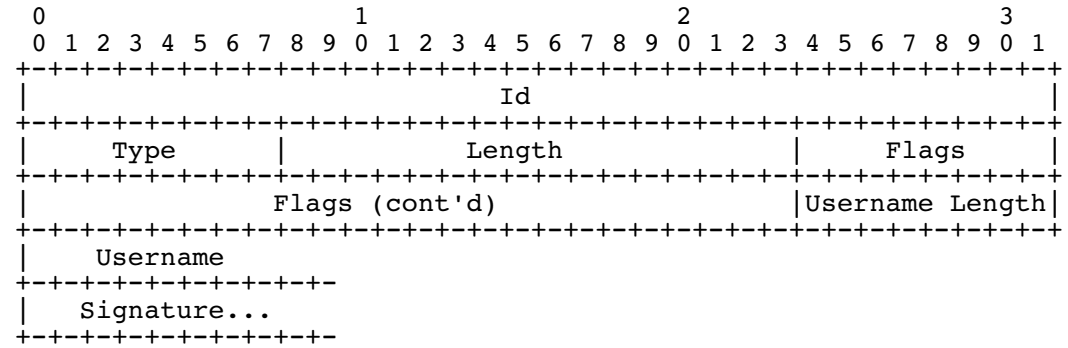
Le champ *Type* indique le type du message. La nature du message est codée dans le premier de poids haut : les valeurs 0 à 127 indiquent des requêtes, les valeurs 128 à 255 indiquent des réponses ou des messages non-solicités.

Le champ *Length* indique la longueur du champ *Body*. Le champ *Body* contient le corps du message, et sa valeur dépend du type de message.

Pour certains messages, le corps peut optionnellement être suivi d'une signature cryptographique (paragraphe 4). Un pair qui n'implémente pas les signatures cryptographiques doit ignorer les données qui suivent le corps.

3.3.1. *Hello* et *HelloReply*

Les relations entre pairs sont maintenues par des messages de type *Hello* = 0 et *HelloReply* = 128 qui ont la structure suivante :



Le corps du message commence par un champ *Flags*, d'une longueur de 4 octets, qui indique les extensions implémentées par le pair (voir partie 5). Un pair qui n'implémente aucune extension met ce champ à 0.

Ce champ est suivi soit du nom d'utilisateur, qui est codé comme un octet de longueur suivi du nom d'utilisateur lui-même. Si le corps contient d'autres données (*Length* est strictement supérieur à *Username Length* + 4), alors les données excédentaires sont ignorées. Le serveur utilise un nom d'utilisateur vide (*Username Length* vaut 0).

Il est *Obligatoire* de répondre à tout message *Hello* par un message *HelloReply* ayant le même *Id*. Il est autorisé d'envoyer un message *Hello* ou un message *HelloReply* non-solicité ($Id = 0$) à tout moment, par exemple pour maintenir un *mapping* dans un NAT.

3.3.2. Transfert de racines

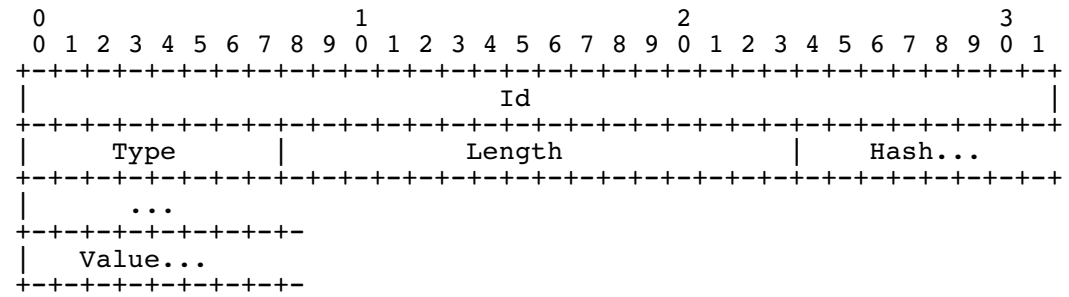
Un pair peut demander le *hash* de la racine d'un autre pair à l'aide d'un message $RootRequest = 1$. Le corps d'un message *RootRequest* est vide (*Length* vaut 0). Un pair qui reçoit un *RootRequest* répond en indiquant sa racine dans un message $Root = 129$. Le corps contient le *hash* de la racine (*Length* vaut 32).

Un pair qui implémente les signatures cryptographiques peut envoyer un message *Root* non-solicité à tout moment. Un pair qui reçoit un message *Root* non-solicité et non-signé doit l'ignorer.

3.3.3. Transfert de données

Un pair peut demander à un pair la valeur d'une donnée en envoyant un message de type $GetDatum = 2$ dont le corps contient un *hash* ($Length = 32$). Si le pair n'a pas la donnée identifiée par le *ash*, il répond par un message $NoDatum = 131$ contenant le *hash* demandé ($Length = 32$).

Si le pair a la donnée identifiée par le *hash*, il répond par un message $Datum = 130$ ayant le format suivant :

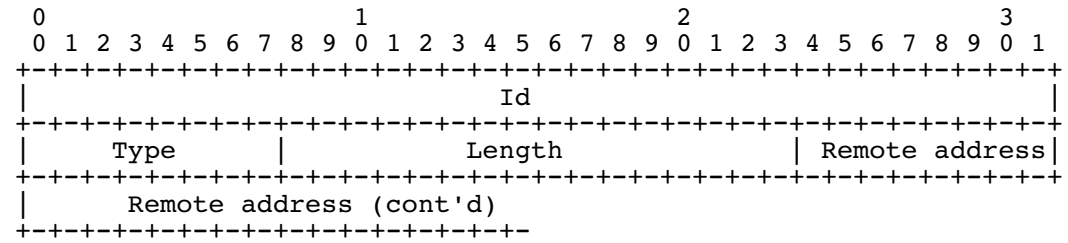


Le champ *Hash* contient le *hash* du champ *Value* ; il a une longueur de 32 octets. Le champ *Value* contient la valeur elle-même, et a une longueur de $Length - 32$ octets.

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*.

3.3.4. Traversée de NAT

Un pair peut demander au serveur de transmettre son adresse de socket à un autre pair. Il le fait à l'aide des messages non-solicités *NAT Traversal Client* et *NAT traversal Server*.



Le champ *Type* vaut 132 pour *NAT Traversal Client* et 133 pour *NAT Traversal server*. Le champ *Remote* address indique l'adresse de socket du client destinataire pour *NAT Traversal Client* et l' adresse du client émetteur pour *NAT Traversal Server* ; il a une longueur de 6 (pour IPv4) ou 18 (pour IPv6) octets.

3.3.5. Message *Error*

Le message *Error* = 254 sert à communiquer les erreurs, et peut servir aussi bien comme réponse que comme message non-solicité. Son corps est une chaîne de caractères lisible par un être humain.

Mon implémentation envoie des messages *Error* amicaux, pensez donc à afficher tous les messages *Error* que vous recevez.

3.4. Autres types de messages

Le mécanisme d'extension (partie 5) permet de déclarer qu'un pair connaît d'autres messages que ceux définis ci-dessus.

4. Signatures cryptographiques

Le protocole peut optionnellement protéger les messages qu'il envoie par des signatures cryptographiques. Les signatures sont générées selon l'algorithme ECDSA sur la courble 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 ;
- une clé publique permet de vérifier les signatures ; les clés publiques sont représentées dans le protocole comme des chaînes de 64 octets (32 octets pour x, 32 octets pour 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. Un pair qui implémente les signatures cryptographiques peut signer tous les messages qu'il envoie, ou alors il peut choisir de ne signer que les messages qui ne sont pas autrement protégés : 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 *Root* et *RootReply* (la racine est le point d'ancrage des arbres de Merkle).

Un pair qui implémente les signatures cryptographiques ignore les messages non-signés qui proviennent d'un pair qui a enregistré une signature auprès du serveur.

Un message *Root* non-solicité est ignoré dans tous les cas s'il n'est pas protégé par une signature correcte. En effet, il est facile d'usurper l'adresse source dans le cas d'un message non-solicité.

Des détails d'implémentation sont donnés à l'annexe A.

5. Mécanisme d'extension

Le protocole contient deux mécanismes qui permettent de l'étendre sans casser la compatibilité avec les autres pairs.

5.1. Types de nœuds

Un pair qui rencontre un type de nœud inconnu (autre que 0 ou 1) le traite comme une feuille de l'abre de Merkle. Ce mécanisme permet à un nœud de publier des données de type autre que texte, par exemple des images ou des fichiers multimédia.

Remarquez que le nœud de type inconnu n'est pas nécessairement une feuille : la structure d'arbre de Merckle permet d'en valider le *hash* sans avoir besoin de comprendre sa structure.

Pour éviter les conflits entre groupes (il n'y a que 254 valeurs disponibles), vous devrez décrire sur la *mailing list* l'extension au protocole que vous comptez implémenter et le ou les numéros de nœuds que vous comptez utiliser.

5.2. Champ *Flags*

Les messages *Hello* et *HelloReply* contiennent un champ *Flags* qui peut servir à indiquer les extensions au protocole implémentées par un pair. Si deux pairs indiquent tous deux une extension dans leurs messages *Hello* et *HelloReply*, alors l'extension est utilisée dans la session qu'ils ont établie.

Pour éviter les conflits entre groupes (il n'y a que 32 valeurs disponibles), vous devrez annoncer sur la *mailing list* l'extension au protocole que vous comptez implémenter et le ou les numéros de nœuds que vous comptez utiliser.

6. Sujet minimal

Au minimum, votre pair devra être capable :

- de s'enregistrer auprès du serveur ;
- de télécharger les messages publiés par un autre pair et de les afficher ;
- de mettre à disposition des autres pairs une liste de messages.

Il est *obligatoire* de vérifier le *hash* d'un nœud avant de s'en servir.

7. Extensions

7.1. Affichage incrémental et mise à jour

Le protocole est conçu pour qu'il soit facile de déterminer rapidement ce qui a changé et ne télécherger que les parties de l'arbre qui ont changé. Il sera apprécié que votre implémentation soit capable de mettre à jour l'affichage de façon efficace lorsqu'un pair publie un nouveau message.

7.2. Traversée de NAT

Le protocole permet de traverser les NAT. La partie passive (répondre à un message *NAT traversal server* est facile, je m'attends donc à ce que vous l'implémentiez. Il est un peu plus difficile de se servir correctement du message *NAT traversal client*.

7.3. *Pipelining* et contrôle de congestion

Je m'attends à ce que votre implémentation soit purement synchrone, et qu'elle ne garde qu'au plus une requête en vol. Si vous implémentez un algorithme plus efficace, il faudra réfléchir aux problèmes de contrôle de flot et de congestion.

7.4. Chiffrage

Le protocole transmet toutes les données en clair. Il serait utile de définir une extension qui permet de chiffrer les données. L'idéal serait de permettre à deux pairs qui implémentent le chiffrage de négocier une clé partagée (par exemple en effectuant un échange Diffie-Hellman sur des courbes elliptiques), puis se servent de la clé symétrique ainsi négociée pour chiffrer les données échangées. (Pensez à utiliser le bon mode.)

7.5. Authentification symétrique

Le protocole d'authentification oblige chaque pair à effectuer une opération sur les courbes elliptiques pour chaque message signé. Il serait plus efficace de négocier une clé partagée, puis de faire de l'authentification symétrique.

7.6. Autres types de données

Le protocole ne définit que des messages textuels. Il serait relativement facile d'étendre le protocole pour pouvoir échanger de petites images. Par contre, pour des images de taille arbitraire, il faudra définir un mécanisme qui permet de stocker de grands objets par morceaux dans l'arbre de Merkle.

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 4 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 = x.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.