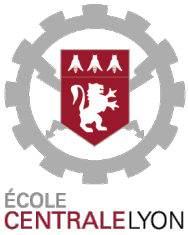


Décentraliser Internet par la fédération : l'exemple d'ActivityPub

Rapport de stage de fin d'étude

du 3 avril au 30 août 2018

Auteure :
Nathalie Rafaralahy
toky.rafaralahy@ecl14.ec-lyon.fr

Tutrice prédagogique :
Marie Goyon
marie.goyon@ec-lyon.fr

Tuteur Framasoft :
Pierre-Yves Gosset
pyg@framasoft.org

Résumé

Les enjeux de la décentralisation d'Internet sont nombreux, les médias sociaux dominants ne respectent pas toujours les libertés des personnes et en font des consommateurs plus que des acteurs des réseaux. Des alternatives aux médias sociaux centralisés existent mais supposent un réel changement de paradigme. Le présent rapport présente les travaux de documentation, de développement logiciel et d'écriture de documentation réalisés lors d'un travail de fin d'étude au sein de l'association Framasoft. Il s'intéresse plus précisément au protocole de fédération ActivityPub, recommandé depuis janvier 2018 par le W3C, principal organisme international de standardisation du web. Ce travail a pour objectif d'enrichir la littérature existante sur les services fédérés. Garantir une multitude et une diversité de documentations, d'articles, accessibles aux non-initiés, aux curieux, aux francophones non-anglophones, aux experts est d'une importance capitale lorsque l'on veut rendre accessible à tout le monde un modèle alternatif.

Mots-clés : *Logiciel libre, protocoles Web, ActivityPub, médias sociaux, alternative, décentralisation, fédération, documentation, preuve de concept*

Abstract

The stakes of Internet decentralization are numerous, the dominant social media do not respect people's freedoms and make them consumers more than network actors. Alternatives to centralized social media exist but imply a real paradigm shift. This report presents the literature search, software development and documentation writing work carried out during an end-of-study work within the non-profit association Framasoft. More specifically, it focuses on the ActivityPub federation protocol, recommended since January 2018 by the W3C, the main international standards organization. This work aims to enrich the existing literature on the *federated web*. Guaranteeing a multitude and diversity of documentation, articles, accessible to the uninitiated, to the curious, to non-English-speaking Francophones, to experts is of capital importance when one wants to make an alternative model accessible to everyone.

Keywords : *Free Software, Web protocols, ActivityPub, social medias, alternative, decentralization, federation, documentation, proof of concept*

Remerciements

Je tiens à remercier tou·te·s les membres de Framasoft de m'avoir donné l'opportunité de travailler dans un cadre aussi chaleureux et bienveillant. Merci pour les moments passés au Framacamp ou aux apéros. Ce fût un plaisir de voir autant de personnalités et de sensibilités différentes réunies par des valeurs communes.

Un grand merci à Pierre-Yves Gosset pour avoir encadré mon stage à la perfection. Merci de m'avoir fait confiance, de m'avoir laisser libre de choisir ce que je voulais faire, d'avoir été à l'écoute et d'avoir toujours su prendre le temps de m'expliquer de nouvelles choses.

Merci à Marie Goyon de m'avoir soutenu dans le choix de ce stage qui peut paraître atypique pour une étudiante de Centrale.

Merci à Chocobozzz d'avoir pris le temps, presque toutes les semaines, de m'expliquer comment ActivityPub fonctionnait et d'avoir été patient quand d'une semaine à l'autre j'avais les mêmes questionnements.

Merci à Mégane et Anne-Marie pour tous les midis, et tous les rires.

Merci à Dorianne et à Chloé, toujours curieuses de savoir sur quoi je travaille. Vous m'avez aidé à me poser les bonnes questions.

Merci à Flo de me pousser, encore et toujours, de me faire croire en moi, petit à petit.

Merci au soutien sans faille de ma famille.

Merci aux personnes qui proposent des imaginaires positifs, qui n'acceptent pas *Black Mirror* comme une fatalité. Merci de continuer à faire vivre les questions de vivre ensemble, de solidarité. Merci de remettre du politique dans ma vie.

Table des matières

Table des figures

Table des listings

Table des sigles

IETF : *Internet Engineering Task Force* - Organisme international de standardisation d'Internet

URI : *Uniform Resource Identifier* - Chaîne de caractère dont la syntaxe respecte une norme et identifiant une ressource sur un réseau.

URL : *Uniform Resource Locator* - Aussi appelée « adresse web », est un sous-ensemble des URI qui identifie les ressources du World Wide Web (document HTML, image, son etc.)

W3C : *World Wide Web Consortium* - Organisme à but non lucratif international principal de standardisation qui promeut la compatibilité des technologies du World Wide Web

Glossaire général

Chemin : *En informatique, le chemin est une chaîne de caractères qui permett de décrire la position d'une ressource. Une adresse web, ou URL, est un chemin vers une ressource Web.*

Données liées : *Représentation des données suivant des standard du Web permettant de les relier entre-elles pour constituer un réseau global d'informations plutôt que des silos de données isolés.*

Format ouvert : *À la différence des formats propriétaires, et dans le même esprit que les logiciels Open Source, ce sont des formats de données dont les spécifications techniques sont publiques, utilisables par tous, sans restriction d'accès ni de mise en œuvre.*

Internet (≠ World Wide Web) : *(Inter Network) est un réseau informatique physique qui relie des ordinateurs entre eux à l'échelle du monde. Il se compose de millions de réseaux publics et privés plus petits (réseaux universitaires, gouvernementaux ou commerciaux). Cette infrastructure repose sur le protocole IP qui achemine les données entre les ordinateurs.*

Instance : *Une instance est une installation d'une application décentralisée sur un serveur. Chaque instance est accessible grâce à une adresse Web spécifique. Il est d'usage de parler d'instance Mastodon ou encore d'instance PeerTube.*

Listing : *lignes de codes d'un programme informatique ou d'un fichier de données dans un format lisible par un humain.*

Protocole (informatique) : *Ensemble de règles qui régissent les échanges de données, d'information entre des éléments connectés, appartennant à un même réseau*

Web social : *Référence à une vision du Web comme espace de socialisation, un lieu où les personnes peuvent intéragir entre elles.*

World Wide Web (≠ Internet) : *Aussi appelé « le Web », c'est la principale application qui repose sur l'Internet, il permet la publication et la consultation de documents multimédias qui sont reliés entre-eux par des liens hypertextes. On y accède grâce à un navigateur web.*

Glossaire spécifique à ActivityPub

Ce glossaire est nécessaire à la compréhension de l'implémentation du protocole ActivityPub.

Activity : En français *activité*, est l'objet fédéré représentant une action réalisée par un *acteur* dans la fédivers. Elle est caractérisée par la personne qui fait l'action (`actor`) et l'objet de cette action (`object`).

Actor : En français *acteur*, est l'objet fédéré représentant un·e utilisateur·ice dans la fédivers. Lorsque l'on parle d'acteur, cela sous-entend toujours « de la fédivers ».

Fédéré - Non fédéré (instances) : Des instances sont dites *fédérées* lorsqu'elles peuvent s'échanger de l'information, les requêtes des unes étant comprises par les autres.

Fédéré - Non fédéré (objet) : Un objet est dit *fédéré* si une copie de cet objet est envoyé, ou mis à disposition, au reste de la fédivers. Un objet est alors *non-fédéré* s'il reste dans la base de donnée de l'instance d'origine et qu'il n'est – ou ne doit être – jamais transféré.

Fédivers : Contraction de *fédération* et *univers* qui désigne l'ensemble des instances fédérées où sont installées des applications implémentant le protocole ActivityPub.

Followers : Autrement dit *les personnes qui suivent l'acteur*, spécifique à chaque *acteur*, désigne le chemin où sont listés tous les acteurs qui souhaitent recevoir les informations partagées par l'acteur.

Following : Autrement dit *les personnes que l'acteur suit*, désigne le chemin où sont listés tous les acteurs dont l'acteur souhaite recevoir les informations partagées.

Inbox : En français *boîte de réception*, spécifique à chaque *acteur*, désigne le chemin où il reçoit toutes les *activités* dont il est destinataire.

Outbox : En français *boîte d'envoi*, spécifique à chaque *acteur*, désigne le chemin où sont listées toutes les *activités* dont il est expéditeur.

User : En français *utilisateur·ice*, est l'objet non-fédéré représentant un·e utilisateur·ice inscrit·e sur l'instance. Cet objet contient les informations de sécurité de l'utilisateur·ice qui lui permettent de s'authentifier sur l'instance.

Introduction

Ce document rapporte le travail réalisé au sein de l'association Framasoft lors du stage de fin d'étude qui s'est déroulé du 3 avril au 30 août 2018. Cette association loi 1901 fait la promotion de la culture libre, le développement et la diffusion de logiciels libres depuis 2004. Après la campagne *Dégooglisons Internet*, l'association souhaite promouvoir d'autres modèles possibles pour les utilisateur·ice·s des services sur Internet.

Cette étude se concentre particulièrement sur le protocole de fédération ActivityPub, standardisé en janvier 2018 par le W3C, organisme international majeur de standardisation du Web. Ce protocole rend possible la fédération de plusieurs plateformes de médias sociaux décentralisés. Cela permet d'agrandir le réseau social tout en gardant une architecture décentralisée des médias sociaux. L'objectif fixé pour ce travail de fin d'étude est de rendre ce protocole plus accessible aux personnes non-initiées mais aussi aux développeur·euse·s qui cherchent à l'implémenter. En effet, si ce protocole est implémenté par de nombreux logiciels, il n'y a que peu de traces de documentation de cette implémentation.

Dans une première partie, une présentation du contexte général ainsi que de l'organisation de ce travail introduisent ce propos. Des bases théoriques sont ensuite posées dans une deuxième partie, pour comprendre l'architecture des médias sociaux sur Internet. La troisième partie est le cœur du travail. Elle fait une présentation technique et non-technique du protocole, du processus de développement d'une preuve de concept ainsi que la documentation de sa réalisation. Les perspectives de ce travail concluront ce dossier.

Partie 1

Contexte et organisation de ce travail

1.1 Contexte et motivations

1.1.1 Problématique de la centralisation sur Internet

L'Internet est un réseau informatique mondial *décentralisé*, accessible au public. On y accède grâce à un fournisseur d'accès internet et il permet d'interconnecter plusieurs réseaux informatiques (publics ou privés) utilisant des protocoles de communication communs.

Si l'Internet est par définition un réseau décentralisé, on observe toutefois une centralisation des services et de l'information vers les « géants du web » aussi appelés GAFAM [1]. Les services proposés par ces entreprises sont massivement utilisés par les internautes et ils jouissent d'un monopole dans leurs domaines respectifs, que ce soit en nombre d'utilisateurs actifs, en nombre de visites ou encore en temps passé sur le service. En 2017, ces entreprises se trouvent dans le top 10 des capitalisations boursières [PwC, 2017] ce qui leur permet, par exemple, de racheter de nouvelles plateformes de médias sociaux pour continuer à asseoir leur monopole.

Les services de médias sociaux de ces entreprises (Facebook, YouTube, Instagram, Twitter...) sont utilisés massivement car ils semblent répondre à un besoin des internautes de faire communauté sur Internet. Ils ne respectent cependant pas leurs libertés : les utilisateurs ne contrôlent pas leurs données personnelles, et le manque de confiance envers ce qu'en font ces entreprises est grandissant (exploitation à des fins publicitaires, marketing, surveillance généralisée, pistage, filtrage de l'information...). De plus, la gouvernance est laissée aux mains de l'entreprise : c'est elle qui décide des contenus autorisés ou non, des algorithmes d'affichage de l'information, et tout cela de manière complètement opaque. L'utilisateur est donc passif dans le fonctionnement de la plateforme qu'il utilise.

1. Pour Google, Apple, Facebook, Amazon, Microsoft

1.1.2 Mouvement pour le logiciel libre

À l'origine, le logiciel libre est défini comme étant un logiciel qui respecte les « 4 libertés » [GNU, 1996] de l'utilisateur :

— La liberté d'utiliser : son utilisation ne doit pas être restreinte
— La liberté d'étudier : son code source doit être ouvert ;
— La liberté de modifier le code ;
— La liberté de diffuser soit une duplication du logiciel soit la version modifiée.

Ces libertés doivent être garanties autant techniquement que légalement. Ainsi, le code source non compilé doit être mis à disposition et le logiciel doit être soumis à une licence dite *libre* (MIT, GPL...).

Certains logiciels les plus utilisés au monde appartiennent à cette catégorie des logiciels libres : Linux, Firefox, Apache ou encore Blender. Ils couvrent des utilisations très diverses : système d'exploitation, navigateur Internet, serveur HTTP ou modélisation 3D.

Aujourd'hui, l'*open source* est en plein essor et le logiciel libre est souvent confondu avec ce dernier. Les débats entre les partisans du *free software* et de l'*open source* ne datent pourtant pas d'hier. Dès 1990, les partisans du logiciel libre défendent une vision de société avec une approche *éthique* de l'informatique alors que les partisans de l'*open source* vantent l'efficacité de cette méthode de développement de logiciels et rejettent toute philosophie [Dachary, 2009, Stallman, 2007].

C'est donc sans surprise que des grandes entreprises du numérique comme Microsoft ou Google se présentent comme les plus importants contributeurs aux logiciels *open source* [Guilloux, 2018], l'efficacité semblant plus attractive et économiquement bénéfique que des considérations éthiques.

Animées par la volonté de construire un Internet plus respectueux de ses utilisateur·ice·s, les personnes qui contribuent au logiciel libre rendent possible l'apparition de nouveaux lieux, sur le Web, qui pourraient permettre aux internautes de se libérer un peu plus du monopole des GAFAM.

1.1.3 L'association Framasoft

Framasoft est une association loi 1901, basée à Lyon, créée en 2004 et initiée dès 2001 par des professeurs de français et de mathématiques voulant créer un annuaire des projets de logiciels libres. Elle compte 35 membres dont 8 salarié·e·s et ne souhaite pas grandir davantage. Une infographie présentant l'association se trouve en annexe A. Son action s'organise, au départ, autour de trois axes principaux : la promotion de la culture libre, la diffusion de services et le développement de logiciels libres. Les valeurs défendues par Framasoft sont [Gosset, 2018] : éthique, émancipation, inclusivité, partage, non-culpabilisation, positivité, pacifisme, faire, bienveillance et soin.

Lors de sa campagne « Dégooglisons Internet » [Framasoft, 2014] qui a duré 3 ans, l'association a mise en place et propose aujourd'hui une cinquantaine de services web libres alternatifs à des services propriétaires utilisés en masse par les internautes (annexe B). Elle s'attache aussi à faire de l'éducation populaire auprès des usagers et usagères

des outils numériques en informant des dangers de la surveillance de masse par les géants d'Internet dont le modèle économique repose sur le pistage et la récupération des données personnelles des internautes.

Figure 1.1 – Illustration des mondes de Contributopia par David Revoy

Cette campagne, terminée en 2017, est suivie par leur nouvelle feuille de route intitulée « Contributopia : dégoogliser ne suffit pas ! » [Framasoft, 2017b]. L'association refuse le modèle de société où l'internaute est simple consommateur passif de services et de contenus proposés par les géants du web. Le projet Contributopia tente de proposer des modèles alternatifs de société où utilisateurs et utilisatrices contribuent ensemble et à leur échelle aux différents projets de la communauté. Le public directement visé par cette campagne sont les associations, l'économie sociale solidaire et écologique, l'éducation populaire.

1.1.4 Développer des alternatives décentralisées

Lors de sa campagne « Dégooglisons Internet », l'association Framasoft a cherché à répondre à la question suivante : « Comment s'émanciper du monopole des GAFAM et reprendre le contrôle sur notre utilisation de l'Internet ? ». Des solutions consistants à décentraliser les services et qui, en plus, respectent nos libertés existent : l'autohébergement de son *Cloud*, de son site Web, de son média social chez soi ou auprès d'associations ou entreprises comme par exemple chez un membre du collectif C.H.A.T.O.N.S. (Collectif des Hébergeurs Associatifs Transparents Ouverts Neutres et Solidaires), initié par Framasoft. Cependant, il est souvent reproché à ces solutions qu'elles n'offrent pas une plateforme unique, permettant potentiellement d'interagir avec des millions de personnes, à l'instar des plateformes centralisées critiquées.

La fédération d'applications est aujourd'hui une solution très prometteuse pour répondre à cette problématique. Voici comment la communauté anglophone de Wikipédia

la définit :

> « A federated social network is an Internet social networking service that is decentralized and distributed across distinct providers (think of email but for social networks). It consists of multiple social websites, where users of each site communicate with users of any of the involved sites. From a societal perspective, one may compare this concept to that of social media being a public utility. »

En français, cela signifie que *la fédération permet aux utilisateurs de différents réseaux sociaux décentralisés* (appelés « instances ») proposant le même service ou bien des services différents, d'échanger de l'information, à condition qu'ils utilisent les mêmes standards pour communiquer. Le réseau décentralisé et fédéré le plus connu est l'e-mail. En effet, bien qu'il existe de nombreux hébergeurs d'e-mails différents, les personnes ayant une adresse e-mail peuvent communiquer entre elles, quels que soient leurs hébergeurs d'e-mail, grâce à des protocoles standardisés.

En juillet 2014, le World Wide Web Consortium (W3C), l'organisme international majeur de standardisation des technologies du World Wide Web, a lancé un programme de développement de standards facilitant l'interopérabilité entre différentes applications [W3C, 2014], i.e permettant à « un système informatique de fonctionner avec d'autres produits ou systèmes informatiques, existants ou futurs, sans restriction d'accès ou de mise en œuvre ».

En janvier 2018, le protocole de fédération ActivityPub devient un standard W3C [W3C, 2018a]. Ce protocole a été implémenté par Mastodon, le média social fédéré le plus utilisé aujourd'hui : il compte plus d'un million de comptes répartis sur presque 1500 instances actives [MNM, 2018]. Il est aussi implémenté par PeerTube, qui se veut une alternative décentralisée et fédérée à YouTube, développée par Framasoft dans le cadre du projet Contributopia. Sa version bêta a été rendue publique en mars [Hermann, 2018] et la sortie de sa première version stable est annoncée pour octobre 2018.

1.2 Définition des objectifs

Le changement de paradigme décrit précédemment soulève de nouvelles questions concernant le fonctionnement et le développement de tels services. Pour de tenter de répondre à ces questionnements, l'objectif de ce travail de fin d'étude est de *vulgariser*[2] *les concepts et les techniques de la fédération des médias sociaux numériques.*

Cette vulgarisation aura deux buts complémentaires, car elle s'adresse à deux publics différents :

1. Permettre à un public plutôt non-initié de comprendre les enjeux de la fédération et comment elle répond aux problématiques liées à la centralisation des services sur Internet ;

2. Action de mettre à la portée du plus grand nombre, des non-spécialistes des connaissances techniques et scientifiques [Larousse, 2018]

2. Permettre à un public plutôt initié de comprendre les intérêts du protocole ActivityPub et rendre sa mise en place plus accessible à des personnes qui voudraient l'implémenter sur leurs logiciels.

1.3 Organisation de ce travail

1.3.1 Acteurs

Ce travail a été encadré
— Par Pierre-Yves Gosset, mon tuteur à Framasoft ;
— Par Marie Goyon, ma tutrice à l'École Centrale de Lyon.
Il a été réalisé grâce à l'aide
— De Florian, développeur de PeerTube à Framasoft ;
— Des membres de Framasoft ;
— De la communauté présente sur Mastodon [3] (via #ActivityPub [4]) et des différents développeurs implémentant ActivityPub dans leurs applications
pour des questionnements plus techniques sur la fédération et son implémentation dans un logiciel avec le protocole ActivityPub.

1.3.2 Moyens et méthodes

À l'instar du travail collaboratif dans lequel s'inscrit la philosophie du logiciel libre, ce travail a constamment été nourri par des échanges avec les acteurs présentés.

Ci-dessous sont listés les principaux moyens et méthodes qui on été mis en œuvre pour atteindre l'objectif visé :
— Découverte du principe de la fédération et du protocole ActivityPub ;
— Veille sur les logiciels implémentant ActivityPub ;
— Rédaction d'articles pour le blog de Framasoft ;
— Création d'un glossaire de la fédération parce qu'elle possède un vocabulaire propre qui n'est pas encore arrêté et qui est essentiellement anglosaxon ;
— Développement d'une preuve de concept : une application simple implémentant ActivityPub ;
— Écriture d'un guide qui sera mis à disposition des développeurs ;
— Participation à la communication autour de la levée de fond pour PeerTube : échange avec un public plus large sur la fédération, aide à la rédaction d'articles

Ce présent document rapporte mon travail mais servira aussi de base à un futur article de tutoriel qui sera publié sur le blog de Framasoft.

1.3.3 Plan du rapport

Ce présent rapport est organisé de la manière suivante

3. Réseau social de micro-blogging libre et décentralisé
4. `https://framapiaf.org/web/timelines/tag/activitypub`

Partie 1 : Présentation du contexte, des objectifs visés et de l'orgnisation de ce travail ;

Partie 2 : Préambule sur l'architecture des médias sociaux sur Internet. Il rend compte de tout le travail de documentation pour comprendre sur quoi se base les services web. Cette partie se veut accessible aux personnes non-initiées aux architectures numériques, elle est nécessaire pour comprendre les enjeux techniques de la décentralisation.

Partie 3 : Cœur de ce travail : l'étude du protocole ActivityPub. Cette partie se compose d'une première section qui a pour but d'expliquer le concept de protocole de fédération sans termes techniques spécifiques. La deuxième section présente les outils qui composent le protocole ActivityPub. Cette présentation est nécessaire pour comprendre le fonctionnement du protocole et faciliter son implémentation dans la preuve de concept réalisée. La troisième section présente le travail de développement d'une preuve de concept et la dernière section est un retour d'expérience sous la forme d'un guide d'implémentation.

Partie 2

Architecture des médias sociaux sur Internet

Cette partie présente les différentes architectures des médias sociaux sur Internet, en définissant et schématisant les termes techniques qui seront utilisés par la suite. Les définitions se veulent simples pour qu'elles soient accessible au plus grand nombre, mais aussi précises pour ne pas trop faire d'approximations.

Si nous nous intéressons à l'infrastructure des médias sociaux dans ce rapport, les définitions qui suivent s'appliquent à tous les services sur l'Internet en général.

Elles peuvent faire débat dans l'étude de l'architecture des systèmes distribués [Bortzmeyer, 2015]. Elles ne sont pas uniques et fixées. Les mots « décentralisé », « acentré », « distribué » peuvent être utilisés dans des contextes différents de celui qui sera présenté par la suite.

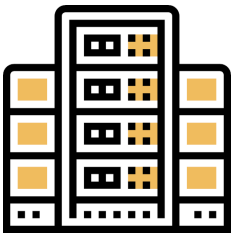
2.1 Définitions

2.1.1 Média social

L'utilisation du terme *média social* plutôt que le terme d'usage, *réseau social*, est très importante. Ce dernier appartient au vocabulaire des sciences humaines et permet de désigner un agencement de lien entre des éléments – individus ou organisations, constituants un groupement (la famille, les collègues, les amis...). Les plateformes utilisées par les personnes pour faciliter les échanges de fichiers et d'information sont *des outils*, des *moyens* (du latin *medium*) au service du réseau social.

Il est important de distinguer les deux termes car leur confusion peut nous amener à croire que le *réseau* est complètement dépendant du *medium*. Le *réseau social* existe indépendamment du *média social* choisi pour l'entretenir.

2.1.2 Serveur - Nœud



Un *serveur informatique* est un ordinateur fonctionnant en permanence (on dit qu'il est *à l'écoute*). Il est connecté au réseau Internet et, comme son nom l'indique, il rend des services aux ordinateur « clients » qui lui font des « requêtes ». Ces services rendus, pouvant être de nature très différentes, constituent en grande partie l'utilisation d'Internet aujourd'hui. La liste suivante présente quelques exemples de serveurs informatiques :

— **Serveur web :** permet au navigateur web d'afficher un site internet ;
— **Serveur de messagerie :** gère la distribution des mails et la possibilité d'y avoir accès depuis n'importe quel ordinateur ;

Un média social est aussi un service informatique. Il sera plus précisément étudié dans les parties suivantes. Dans le cadre de cette étude, nous pourrons aussi appeler *nœud du réseau* un serveur du réseau informatique. En effet quelle que soit la structure étudiée, à l'exception des structures distribuées ou en pair-à-pair (section 2.3.2.), les échanges de fichiers et d'informations passent par un serveur.

Par la suite, l'infrastructure physique et le service proposés seront confondus lorsqu'ils sont détenus par une même entité. Ils seront appelés *service*.

2.1.3 Client - Acteur

En règle générale, le *client informatique* est l'application web, mobile ou bureau avec laquelle l'utilisateur interagit. Le client gère l'affichage de l'information donnée par le serveur afin qu'elle soit lisible par l'utilisateur. La liste suivante présente des exemples de clients aux services présentés précédemment :

— **Client web :** c'est le navigateur web qui est utilisé (Firefox, Chrome, etc) pour consulter un site web ;
— **Client de messagerie :** c'est l'application qui permet d'accéder à la messagerie. À l'École Centrale de Lyon, le client de messagerie principal est une interface fournie par Zimbra. Thunderbird est aussi un client de messagerie qui est installé directement sur l'ordinateur de l'utilisateur ;

Par extension, on désigne souvent comme « client » l'ordinateur sur lequel est installé l'application. Cependant, il ne faut pas confondre l'*application client* et *l'ordinateur client*. Par exemple, il existe un nombre limité de navigateurs web (Firefox, Chrome, etc) par contre, ces applications sont installées sur des millions d'ordinateurs ou de téléphones. De la même manière, l'application de messagerie *WhatsApp* n'a qu'une unique *application* client sur téléphone mobile mais elle est installée sur des millions de téléphones. Ce sont ces millions d'ordinateurs qui seront qualifiés de *clients* dans le cadre de cette étude. Ils

seront aussi appelés *acteurs* car ils sont une *représentation informatique* des utilisateurs. Cet aspect sera plus détaillé dans la partie 3.

En pratique, plusieurs applications clients de médias sociaux différents peuvent être installés sur un même ordinateur. Dans le cadre de l'étude, si deux applications clients de deux médias sociaux différents sont installés sur un même ordinateur, ils seront considéré, sauf mention contraire, comme deux *acteurs* différents.

2.1.4 Requête/Réponse dans un environnement client-serveur

Un environnement *client-serveur* est un mode de communication à travers l'Internet entre les clients et les serveurs. Le client envoie des « requêtes » au serveur et celui-ci renvoie des « réponses » (figure 2.1). Un même serveur fait généralement l'objet de requêtes d'un très grand nombre de clients.

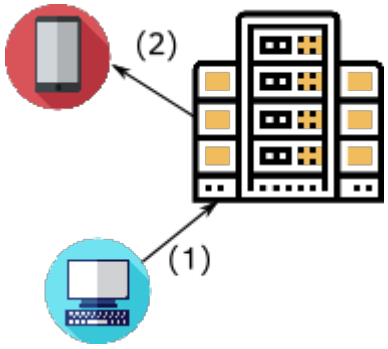
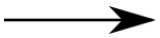


Figure 2.1 – Schéma d'un requête d'un client vers le serveur (1) et d'une réponse du serveur vers un client (2)



On appelle requête *un message* envoyé par un client au serveur. Une requête entraîne toujours une réponse du serveur même si c'est un message d'erreur. Les requêtes et réponses client-serveur seront par la suite représentés par la flèche continue ci-contre.

Les requêtes peuvent prendre des formes différentes, et le langage dans lequel ils sont écrits n'est pas forcément connu des utilisateurs. Voici des exemples de requêtes qui pourraient être envoyées traduites en langage humain ainsi que des réponses possibles :

- Requête à un serveur Web : « Récupère le code `HTML` de la page web `https://framasoft.org` » ;
  Réponse : envoi du code HTML demandé ;
- Requête un serveur de messagerie : « Envoie ce mail à ; `narf@framasoft.org` » ;
  Réponse : notification de l'envoi réussi ou d'une erreur ;
- À un serveur applicatif : « Je t'envoie mes identifants, vérifie qu'ils sont bons et connecte-moi à ce jeu en ligne » ;
  Réponse : identifiants vérifiés et connexion au jeu.

2.1.5 Requête/Réponse dans un environnement serveur-serveur

Un environnement *Serveur-Serveur* est un mode de communication à travers l'Internet entre plusieurs serveurs. Deux serveurs peuvent se s'envoyer des requêtes et se répondre (2.2).

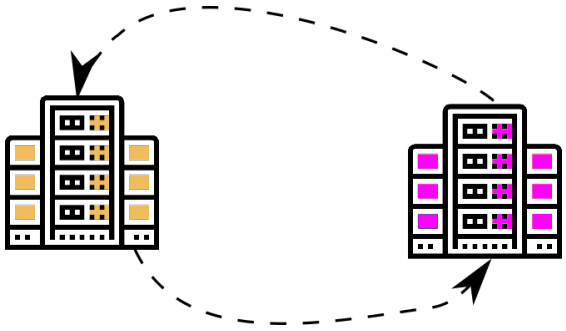
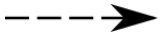


Figure 2.2 – Schéma de requêtes entre deux serveurs



Les messages échangés entre serveurs sont représentés par la flèche pointillée ci-contre. De manière générale, le langage utilisé par les serveurs pour communiquer entre eux n'est pas le même que celui utilisé entre un client et le même serveur. Ce langage peut être libre, i.e. il est défini précisément et utilisable par d'autres, ou propriétaire, dans ce cas seuls les deux serveurs se comprennent.

2.2 Média social centralisé

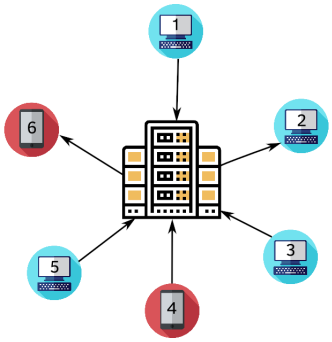


Figure 2.3 – Schéma d'un service centralisé

Un média social est dit *centralisé* lorsque son intégrité dépend d'une seule entité sans laquelle il ne peut fonctionner. Les médias sociaux les plus populaires aujourd'hui fonctionnent sur ce principe (Facebook, YouTube, Twitter, Reddit, pour n'en citer que

quelques uns). Leur taille nécessite que leur infrastructure repose sur de nombreux serveurs physiques mais les entreprises les gèrent comme un unique service. Ainsi, comme schématisé sur la figure 2.3, les clients n'effectuent de requêtes qu'à un unique service. Si (1) veut contacter (6), alors il envoie une requête au service avec le message à transférer à (6) et le service poste le message à (6) qui le reçoit dans sa boîte de réception.

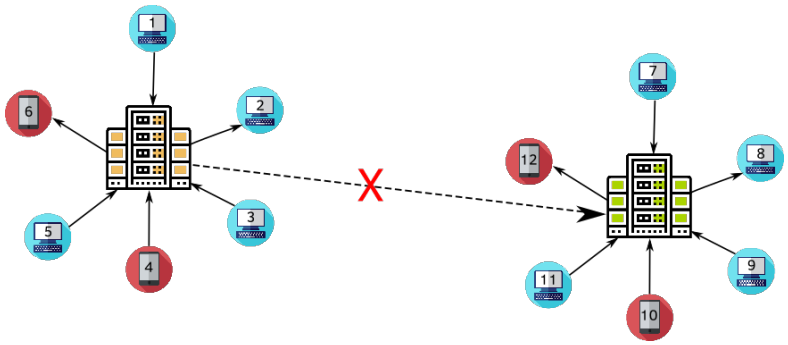


Figure 2.4 – Schéma de deux services centralisés

Les services centralisés sont beaucoup plus simple à réaliser et à gérer sur le plan technique : la méthode d'intégration verticale permet a un unique pôle de décider de l'évolution de la plateforme. Toutes les informations sont facilement trouvables car elles se situent au même endroit, la solution technique est uniforme et généralement il n'y a que peu d'*applications clients* associés, eux aussi souvent développés par la même entreprise. Les problèmes que l'on rencontre sur ce type de plateformes ont déjà été évoquées précédemment :

— Un seul service doit supporter une montée en charge, une panne, une attaque et s'il ne la supporte pas, tout le service peut tomber en panne et ne plus être accessible par les clients [1] ;
— L'entreprise peut récolter les données personnelles des utilisateurs en masse et peut censurer les contenus publiés ;
— Deux médias sociaux centralisés ne peuvent pas communiquer entre eux : on dit que ce sont des *Walled Garden* [Memetic, 2012], en français « jardin muré ». Ainsi sur la figure 2.4, le client (1) ne peut pas envoyer un message au client (10).De cette manière, un utilisateur (u1) de Facebook ne peut pas envoyer un message à un utilisateur (u2) de Twitter. (u1) doit s'inscrire sur Twitter pour contacter (u2), il utilisera alors **deux clients différents**, un pour échanger avec les utilisateurs de Facebook et un autre pour échanger avec ceux de Twitter.

Dans ce modèle, on dit que l'intelligence est au centre du réseau.

1. Les anglophones appelle cela un SPOF (Single Point Of Failure) soit un point unique de défaillance

2.3 Média social décentralisé

À l'opposé d'un média social centralisé, un média social *décentralisé* signifie qu'il n'y plus d'autorité unique et centrale qui contrôle tout le réseau. Dans ce cas, plusieurs nœuds du réseau peuvent proposer un même service. Dans un modèle décentralisé, ces nœud sont aussi appelé *instances*. Ce modèle permet d'éviter des problèmes rencontrés par la centralisation :

- On évite la montée en charge anormale des services car le même service est proposé par plusieurs entités ; il n'est plus nécessaire d'investir dans une grosse infrastructure coûteuse comme un *datacenter* par exemple. Comme les services sont moins chargés, ils n'ont pas à gérer un aussi grand nombre de données des utilisateurs ;
- Si un service n'est plus accessible, le reste du réseau n'est pas impacté : le réseau est plus résilient ;
- Si un utilisateur n'est pas satisfait de la gestion de l'instance sur laquelle il est inscrit (modération, censure, non respect des libertés...), il peut toujours en changer. En pratique, la migration peut toutefois être plus ou moins pénible car les données ne peuvent pas toujoours migrer avec l'utilisateur ;
- Deux médias sociaux décentralisés de nature différente (exemple : micro-blogging et vidéos) peuvent potentiellement communiquer entre eux à condition de définir un langage commun ;
- Chaque nœud du réseau, ou *instance*, peut définir ses propres règles et conditions d'utilisation notamment si le nœud regroupe plusieurs utilisateurs (cf. 2.3.1).

Les médias sociaux décentralisés sont un réel changement de paradigme autant pour les développeurs que pour les utilisateurs. Si l'Internet est décentralisé par essence, l'usage qu'utilisateurs et développeurs en font, pour une grande majorité, est celle des modèles centralisés. La partie 3 propose un aperçu de la complexité de ce nouveau paradigme, notamment causée par un nouveau vocabulaire, un fonctionnement différent, le grand nombre de protocoles utilisés, et la flexibilité de ces protocoles.

Ainsi, les personnes habituées aux médias sociaux centralisés doivent fournir un effort de compréhension car le système est plus complexe, il n'y a plus une unique porte d'entrée, toute l'information ne se trouve pas au même endroit, et il existe un vocabulaire spécifique souvent technique.

De la même manière, les personnes développant ce type de logiciels doivent s'accorder sur les protocoles à mettre en oeuvre, comprendre et appliquer les nombreux protocoles existant, dans le but de faire vivre et faire grandir le réseau. Dans le cas d'un média social « isolé » – et encore plus lorsque l'application de base n'est pas libre, il n'y a pas ce genre de questionnements car les protocoles sont internes, souvent propriétaires, et il n'y a aucune volonté à s'ouvrir à un réseau plus grand dont ils ne maîtrisent pas la gouvernance. Le système décentralisé se complique d'autant plus que parfois, de simples différences de versions des logiciels utilisés empêchent des services de communiquer entre eux.

Les avantages que les médias sociaux décentralisés apportent vis à vis du respect des libertés, de la vie privée des personnes, de la reprise de contrôle sur le fonctionnement de l'Internet, de la résilience et la diversité du réseau sont pourtant nombreux, ce qui pousse à leur étude. Les services décentralisés regroupent les services fédérés et distribués.

2.3.1 Média social fédéré

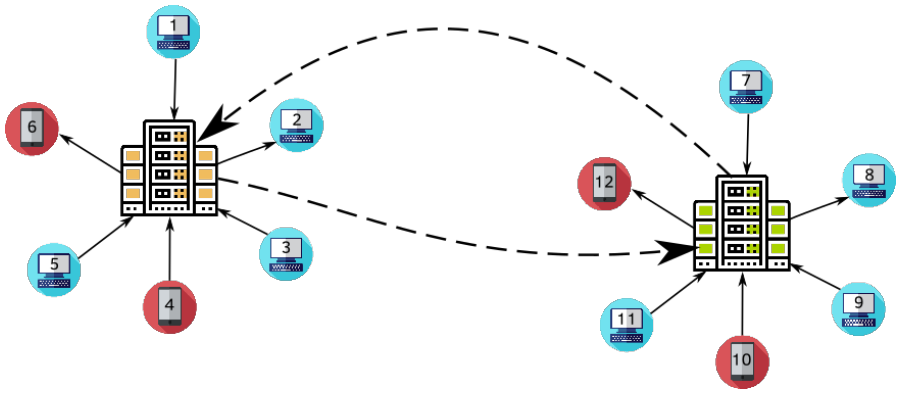


Figure 2.5 – Schéma de deux services fédérés

Un média social fédéré est un média social décentralisé dans lequel nous pouvons retrouver la structure habituelle des services centralisés où des clients qui requêtent un unique serveur, mais cette fois-ci les serveurs fédérés peuvent communiquer entre eux, comme l'illustre la figure 2.5.

Les services qui communiquent entre eux sont dit « fédérés », chaque serveur est accessible par sa propre URL. Les services peuvent être :

1. De même nature : le même logiciel qui est installé sur deux serveurs différents
2. De nature différente : des logiciels différents qui sont installés

Cette communication suppose l'utilisation d'un langage commun, défini entre les deux serveurs.

Dans le premier cas, il n'est pas forcément nécessaire que le langage de communication suive des normes publiques. Généralement le développement d'un logiciel se fait par un nombre restreint de personnes. Si le logiciel est libre alors le protocole de communication est ouvert mais pas forcément normalisé.

Dans le second cas, il est nécessaire que le langage de fédération soit ouvert et normalisé : si des logiciels différents développés par des personnes différentes souhaitent communiquer, chacun doit respecter un certaine nombre de règles pour se comprendre. Normaliser et utiliser un protocole d'échange ouvert permet alors à d'autres acteurs d'agrandir le réseau librement.

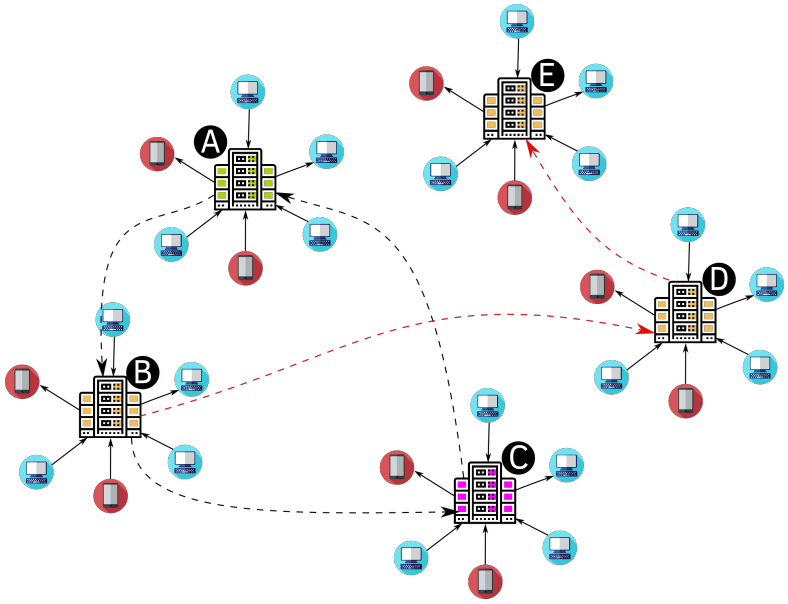


Figure 2.6 – Schéma d'un réseau fédéré

Un service peut implémenter plusieurs langages de fédération. Il en existe de nombreux, pour des usages différents et ils peuvent aussi évoluer. Ainsi, sur la figure 2.6, on peut voir deux réseaux fédérés, le réseau noir (serveurs A, B et C) et le réseau rouge (serveurs B, D et E). Le réseau rouge relie des serveurs implémentant le langage de fédération *rouge* et le réseau noir celles implémentant le langage *noir*.

— Le service B implémente les deux langages de fédérations rouge et noir. À l'instant t, il envoie un message aux services D et C et il reçoit un message du service A. À un instant $t + 1$, il pourrait potentiellement échanger avec E ;
— Un client de A peut échanger des messages avec un client de B ou de C mais pas de D ni de E.

Par ailleurs, des mécanismes de blocages et de connexion unilatérale peuvent être mis en place. Ils seront étudiés en détail dans la partie 3.

Dans ce modèle, on dit que l'intelligence est à la périphérie du réseau.

2.3.2 Média social distribué (ou pair-à-pair)

Un service distribué est un service décentralisé où chaque entité du réseau est à la fois un client et un serveur (figure 2.7) : c'est le modèle du pair-à-pair (en anglais *peer-*

to-peer) qui est utiisé par des systèmes comme le BitTorrent [2], ou encore la Blockchain [3], pour ne citer que les plus connus.

Ce modèle est encore plus robuste car il n'y a absolument plus aucun pouvoir central. Chaque nœud est indépendant et aucun n'est nécessaire au fonctionnement du réseau. Sa réalisation se retrouve face aux mêmes difficultés que tout média social décentralisé mais à une échelle beaucoup plus grande car l'implémentation se fait au niveau de chaque utilisateurs.

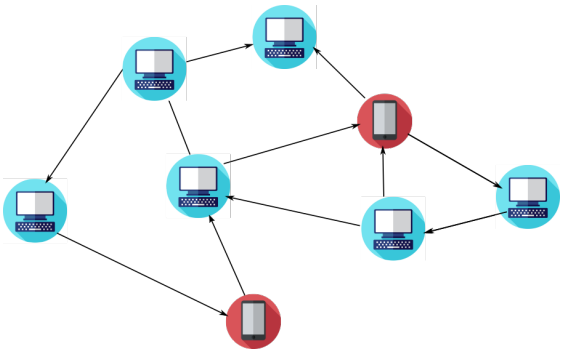


Figure 2.7 – Schéma d'un réseau pair-à-pair (`peer-to-peer`)

Dans ce modèle, l'intelligence est à chaque nœud du réseau.

2. Protocole de partage de fichiers en pair-à-pair. Souvent connu pour son utilisation pour le partage de fichiers illégaux (films, musique), il n'est pourtant pas illégal. Il est permet de partager des fichiers efficacement entre plusieurs personnes qui ne disposent pas forcément toutes d'une bonne capacité de téléversement. Il peut être utilisé en entreprise, ou pour le téléchargement de fichiers volumineux tels que des archives ou des logiciels.

3. Technologie de stockage et de transmission d'information. Cette base de donnée est dite distribuée car tous les pairs du réseau en détiennent une copie.

Partie 3

Le protocole de fédération ActivityPub

Cette partie est composée de deux sections. La première section a pour but d'expliquer le concept de protocole de fédération avec peu de termes techniques. La deuxième section, plus technique, présente les différents outils qui composent le protocole AcitivityPub. L'implémentation de ce dernier dans la preuve de concept réalisée pour cette étude sera présentée sous la forme d'un guide.

3.1 Des fédérations sur l'Internet

Cette section s'inspire du travail de Nathanaël Jourdane avec son accord [Jourdane, 2018].

Lors de la campagne *Dégooglisons Internet* [Framasoft, 2014], certains services alternatifs proposés par Framasoft ont eu plus de succès que d'autres. Si choisir des services décentralisés permettent de se libérer des géants du Web, certains changements restent plus évidents à faire. Ainsi, un service comme *Framadate*, qui permet de planifier des rendez-vous et faire des sondages, peut facilement être adopté par le grand public, même peu sensible aux valeurs du libre. Il s'utilise sans inscription au préalable et la création du sondage peut être à l'initiative d'une seule personne : l'utilisation est plutôt simple et personne ne s'engage réellement envers la plateforme.

Pour les services de médias sociaux, le changement est plus compliqué car il faut pouvoir contrer l'*effet de réseau* engendré par les services centralisés. Autrement dit, peu de personnes sont prêtes à quitter les médias sociaux centralisés pour des médias sociaux décentralisés car les premiers sont fréquentés en général par une grande partie de leurs connaissances (amis, famille, travail, public pour les créateur·ice·s), et les seconds sont moins visibles, les instances ne sont pas faites pour supporter une aussi grande charge d'utilisateur·ice·s que les premiers.

Comment faire pour que ces médias sociaux décentralisés se lient, interagissent, sans s'unifier ?

Figure 3.1 – Carte de la Gaule avant la Guerre des Gaules selon Gustav Droysen

3.1.1 Qu'est ce que « fédérer » ?

La campagne *Dégooglisons Internet* prenait pour métaphore des camps gaulois – les logiciels libres – qui luttent contre l'invasion romaine – les logiciels propriétaires (annexe B). Si l'on reprend cette métaphore, il est important de préciser que la Gaule n'était pas un pays unifié. Il s'agirait alors de parler *des peuples* gaulois qui occupaient les différentes parties du territoire, qui avaient leurs propres cultures et patois locaux (figure 3.1). Ces groupes sont comparables à différentes instances de médias sociaux décentralisés : ce ne sont pas toujours les mêmes logiciels, ce ne sont pas les mêmes centres d'intérêts etc.

Le peuple romain, lui, est un peuple unifié. L'armée est grande et dirigée par un homme ambitieux qui souhaite élargir son territoire quitte à sacrifier les populations locales et leurs cultures. Elle est comparable aux plateformes de médias sociaux propriétaires élargissant leur territoire en achetant d'autres plateformes mais aussi en multipliant leurs fonctionnalités en fonction de ce qu'elles identifient comme le *besoin général*, à lissant ainsi les différences entre les individus et leurs cultures.

Comment alors lutter contre cette invasion ? Fédérer, du latin *foederare* qui signifie « unir », « rapprocher par une alliance », trouve deux sens dans le dictionnaire [Wiktionnaire, 2018] :

1. Réunir en fédération ;
2. Regrouper des personnes dans un but commun.

Dans le cas étudié, ces deux définitions ont du sens. La première définition peut s'interpréter comme le *moyen* de se fédérer et la deuxième permet de poser la question du *but commun* que la fédération cherche à atteindre.

Si l'on reprend la métaphore précédente, le but commun des peuples gaulois est clair : se défendre de l'invasion romaine.

Il est possible d'imaginer de nombreux moyens pour atteindre ce but. Cependant, il ne faut pas croire que tous les moyens sont bons. Les peuples gaulois ne veulent pas sacrifier leurs différentes cultures, ce qui est compréhensible : elles définissent leur groupe. De plus, rien ne dit qu'ils ont des affinités entre peuples : ils ont, tout au plus, un ennemi commun. Leurs différences de culture peut aussi être une réelle force, chaque peuple ayant ses propres spécialités en fonction de son territoire : les romains ne peuvent pas espérer tous les battre de la même manière.

Pour se protéger sans avoir à s'unifier, pour s'organiser au mieux, se tenir informés, il faut pouvoir améliorer l'entente entre les différents peuples. Or, si ces peuples ne parlent pas la même langue, cela risque d'être encore plus difficile. La méthode pour se *réunir en fédération* doit donc passer par la recherche d'un moyen de communication et d'interaction.

3.1.2 Un langage commun

Pour fédérer, il est nécessaire de pouvoir communiquer. Créer une langue commune pour faciliter les échanges d'informations, les échanges marchands, ou pour se lier d'amitié entre différents groupes, n'est pas une idée nouvelle. Il est important que cette langue commune puisse être comprise de tous : elle ne doit pas se substituer aux différents patois, elle doit rester malléable pour s'adapter aux besoins à exprimer et doit faire en sorte de ne pas effacer les subtilités de ce qui peut être exprimé dans les différents patois.

Il semble que construire un tel langage et le rendre accessible et compréhensible par tout le monde n'est pas simple. Sur Internet, ces langages sont appelés *protocoles*. Leur standardisation permet leur diffusion et lorsque ce sont des protocoles *ouverts*, comme ActivityPub, ils peuvent être modifiés et rediffusés afin de correspondre au mieux aux besoins des utilisateur·ice·s.

ActivityPub n'est pas le premier langage de fédération, et ne sera sans doute pas le dernier. Il est très utilisé dernièrement car il a été recommandé par un organisme international majeur de standardisation, le W3C.

Il permet aux différentes plateformes de médias sociaux de se lier dans le grand réseau appelé la *fédivers* – mot-valise pour fédération et univers. Il existe des milliers de d'instances différentes sur le réseau, on y trouve :

— Une instance de micro-blogging généraliste : `https://framapiaf.org` ;
— Une instance de micro-blogging spécialisée dans la photographie : `https://photog.social` ;
— Une instance de partage de vidéos spécialisée dans la zététique : `https://skeptikon.fr` ;
— Une instance de partage de musiques libres : `https://open.audio` ;
— Et plein d'autres...

Cette multitude d'instances signifie qu'il y a une multitude de portes d'entrée pour accéder à la fédivers : le choix de la plateforme sur laquelle s'inscrire ne se fait plus uniquement par rapport aux plateformes choisies par nos connaissances, comme c'est souvent le cas pour les médias sociaux centralisés. Ce choix peut se faire en fonction de très nombreux paramètres :

Figure 3.2 – Logo du protocole ActivityPub

— Les centres d'intérêts : politique, technologie, animaux, littérature, art, musique libre... ;
— Le type de contenu que l'on souhaite partager en priorité : texte, musique, vidéo, photo... ;
— Les valeurs : liberté d'expression, solidarité... ;
— les conditions d'utilisation de l'instance ;
— etc.

Grâce au protocole de fédération, ce choix d'instance reste personnel et ne ferme pas la porte au reste du réseau : une personne inscrite sur une instance spécialisée en photographie peut être amie avec une autre personne inscrite sur une instance spécialisée en politique.

Il faut toutefois noter qu'au vu de la jeunesse du protocole, si le nombre de plateformes offrant des services différents sur la fédivers est en hausse, elles ne sont pas encore toutes entièrement compatibles entre elles. Le travail réalisé lors de ce stage de fin d'étude permet d'enrichir les sources de documentations pour une meilleure la connaissance du protocole.

3.2 Les outils techniques à l'origine du protocole ActivityPub

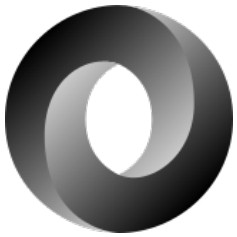
3.2.1 Une recommandation du W3C

Le protocole ActivityPub est un standard proposé par le *Social Web Working Group* du W3C. Il naît d'un programme lancé par le W3C en juillet 2014 qui visait à développer des standards facilitant l'interopérabilité des applications du Web social. Il acquiert le statut de *W3C recommandation* en janvier 2018.

Ce protocole se base sur de nombreux standards recommandés par le W3C ou l'IETF.

3.2.2 JSON et JSON-LD

Dans le protocole ActivityPub, le format `JSON-LD` est utilisé pour représenter les `objets`. Pour comprendre ce format, il est d'abord nécessaire de présenter le format `JSON`, sur lequel il se base.



JavaScript Object Notation(`JSON`) n'est pas un langage de programmation mais un mode de représentation de données textuelles structurées, dérivé de la notation des objets en langage JavaScript. C'est un format ouvert qui représente l'information de telle sorte qu'elle soit facile à écrire et à lire pour un humain. Il doit aussi pouvoir être généré et interprété par un ordinateur. Il peut être utilisé indépendamment du langage utilisé pour la programmation.

Ses conventions d'écritures sont similaires à celles de langages descendants du C. Un document au format `JSON` a une extension en `.json`.

Un document `JSON`, délimité par des accolades (`{ }`), représente un objet ou un tableau et comprends deux types d'éléments :

— Des couples « `"clé"`/`"valeur"` » liés par « `:` » : exemple le couple `clé_1/valeur_1` (Listing 3.1, ligne 2) ;

— Des listes ordonnées de valeurs délimitées par des crochets `[valeur 1,valeur 2]`) : exemple la valeur associée à `clé_2` (Listing 3.1, lignes 3-6)

```
1  {
2    "clé_1" : "valeur_1",
3    "clé_2" : [
4      "Première valeur de la liste",
5      "Deuxième valeur de la liste"
6    ],
7    "clé_3" : {
8      "sous_clé_1" : "sous_valeur_1",
9      "sous_clé_2" : "sous_valeur_2"
10   }
11 }
```

Listing 3.1 – Exemple d'un document `JSON`

Les `valeurs` peuvent être un autre objet (Listing 3.1, lignes 7-10), un tableau, un booléen, un nombre, une chaîne de caractère ou `null`, utilisé lorsqu'il y a une absence intentionnelle de valeur.

Le listing 3.2 donne l'exemple d'un objet `JSON` représentant l'auteure, et peut se lire ainsi :

> « L'objet présenté se nomme Nathalie. On la surnomme narf. Elle a 25 ans. Elle habite à Lyon. Elle est étudiante à l'École Centrale de Lyon et à l'Université Lyon 3. Elle connait une personne dont le nom est Pierre-Yves. Son nom de famille est Gosset. Il habite à Lyon. »

Ce mode de représentation de données est donc facilement lisible par une personne.

```
{
  "name": "Nathalie",
  "nickname":"narf",
  "age": "25",
  "city": "Lyon",
  "school": [
     "École Centrale de Lyon",
     "Université Lyon 3",
    ],
  "knows": {
     "name" : "Pierre-Yves",
     "surname" : "Gosset",
     "city" : "Lyon"
    }
}
```

Listing 3.2 – Exemple d'objet en format `JSON` représentant l'auteure

Cependant, les humains interprètent les données qui leur sont présentés. L'entrée « `"city" : "Lyon"` » peut effectivement signifier « Nathalie vit à Lyon », mais elle aurait aussi pu vouloir dire « Nathalie est née à Lyon » ou encore « Lyon est la ville préférée de Nathalie ». Si la machine peut lire les couples clés/valeurs, elle n'a pas la capacité d'interprétation humaine. Or, le protocole ActivityPub a pour but d'échanger des informations entre différentes machines : le mode de formatage des données qu'il utilise doit donc normaliser la définition des clés et des valeurs pour être sûr qu'il n'y ait pas d'interprétation faussée d'une machine à l'autre.

Une telle normalisation est possible grâce à la construction de vocabulaires adaptés aux usages des applications. Ils fonctionnent comme des dictionnaires définissant précisément l'usage des clés. Le mode de formatage de données basé sur le `JSON`, qui prend en compte les problématiques énoncées précédemment et qui est utilisé par le protocole ActivityPub est le `JSON-LD`.

JavaScript Object Notation for Linked Data (`JSON-LD`) est un mode de formatage de données basé sur le `JSON` qui donne un moyen simple aux développeurs de transformer des données existantes en `JSON` vers du `JSON-LD`. C'est une recommandation du W3C, il est donc considéré comme un standard du Web.

Les *données liées*, aussi appelées *Web des données* [Berners-Lee, 2001], selon une définition du W3C, répondent aux problématiques présentées précédemment : les données ne transitent plus de manière isolées les unes des autres mais sont reliées entre elles pour former un réseau global d'information. Les données liées suivent alors le modèle suivant :

Le sujet : il représente la ressource à décrire ;

Le prédicat : il représente une propriété applicable au sujet, qui doit appartenir à un vocabulaire ;

L'objet : il correspond à la `valeur` de l'objet `JSON`.

L'exemple (B) de la figure 3.3 permet donc de voir que l'objet `Pierre-Yves` peut être aussi être un sujet. Le prédicat `vit à` est toujours compris de la même manière. Les objets `Lyon` et `France` sont chacun interprétés de la même manière.

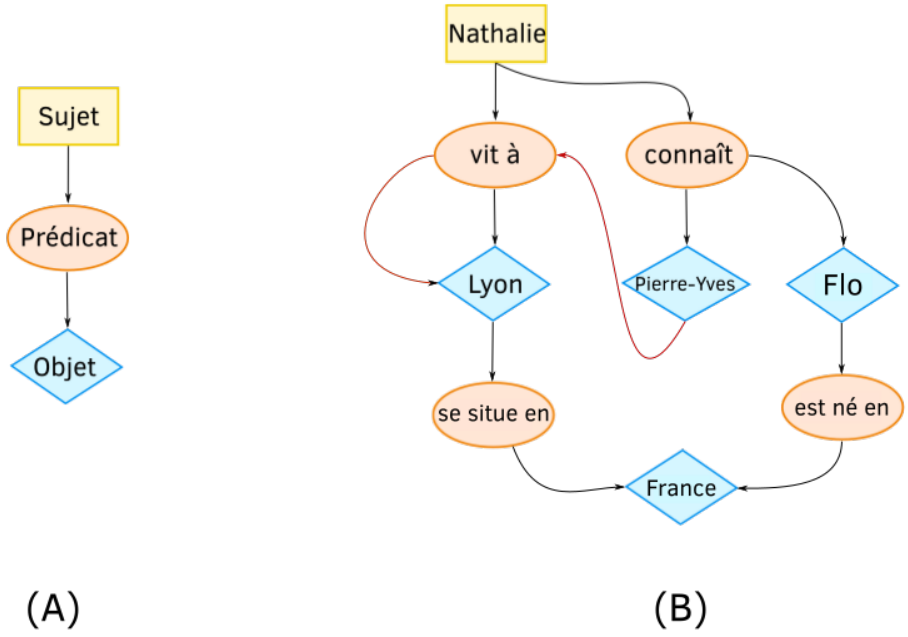


Figure 3.3 – (A) Triplet RDF | (B) Exemple simple de données liées

Le Web des données s'appuie lui aussi sur d'autres standards pour uniformiser la manière dont un vocabulaire est mentionné. Il repose sur quatre piliers dont l'énonciation peut être simplifiée ainsi :

1. Chaque élément doit avoir par un identifiant unique (URI [**?**]) ;
2. Les identifiants utilisés sont des adresses web (URL, HTTP) ;
3. Des informations exploitables par l'humain **et** par la machine doivent se trouver à cette adresse web ;
4. L'identifiant est relié à d'autres identifiants afin d'améliorer la découverte d'information sur le Web.

Il existe plusieurs vocabulaires utilisés pour les données liées, qui sont adaptés à des utilisations différentes. En voici quelques exemples :

FOAF : Friend Of A Friend

— Identifiant : `http://xmlns.com/foaf/0.1/`
— Décrit les personnes et les relations qu'elles entretiennent entre elles

ActivityStreams :
— Identifiant : `https://www.w3.org/ns/activitystreams`
— Décrit des activités de personnes sur les médias sociaux
— Utilisé par ActivityPub, et sera développé dans la sous-partie 3.2.3

DBpedia :
— Identifiant : `http://dbpedia.org/page/`
— Propose une version structurée et normalisée au format du web sémantique des contenus de Wikipedia

Le listing 3.3 est la version en `JSON-LD` de l'objet du listing 3.2. Le premier étant moins lisible que le second, il existe une autre manière de présenter des données liées, en utilisant la clé « `"@context"` » qui aura pour valeur une liste des différents vocabulaires utilisés comme présenté dans le listing 3.4.

```
{
  "http://xmlns.com/foaf/0.1/givenname": "Nathalie",
  "http://xmlns.com/foaf/0.1/nick": "narf",
  "http://xmlns.com/foaf/0.1/age": "25",
  "http://xmlns.com/foaf/0.1/based_near": "http://dbpedia.org/page/Lyon",
  "http://xmlns.com/foaf/0.1/schoolHomepage": [
     "https://ec-lyon.fr/",
     "https://univ-lyon3.fr/"
   ],
  "http://xmlns.com/foaf/0.1/knows": {
     "http://xmlns.com/foaf/0.1/name" : "Pierre-Yves",
     "http://xmlns.com/foaf/0.1/familyName" : "Gosset",
     "http://xmlns.com/foaf/0.1/based_near": "http://dbpedia.org/page/Lyon"
   }
}
```

Listing 3.3 – Objet précédent en données liées (`JSON-LD`)

```
{
  "@context": [
     "http://xmlns.com/foaf/0.1/",
     "http://dbpedia.org/page/"
    ],
  "givenName": "Nathalie",
  "nick": "narf",
  "age": "25",
  "based_near": "Lyon",
  "schoolHomepage": [
     "https://ec-lyon.fr/",
     "https://univ-lyon3.fr/"
    ],
  "knows": {
     "name" : "Pierre-Yves",
     "familyName" : "Gosset",
     "based_near" : "Lyon"
    }
}
```

Listing 3.4 – Objet précédent en données liées avec contextualisation

3.2.3 ActivityStreams

Activity Streams [W3C, 2011] est le vocabulaire ayant inspiré le protocole ActivityPub et sur lequel ce dernier se base. Ce format ouvert, standardisé par le W3C, alimentant le format `JSON-LD`, propose une manière de représenter les différentes activités sociales pouvant avoir lieu sur le Web.

Ce vocabulaire définit les objets interagissant sur le Web social ainsi que les types d'interaction qu'il y peut exister. L'avantage du format ouvert de ce vocabulaire est qu'il est possible de le faire évoluer, les objets et les interactions pouvant changer avec le temps. Il faut toutefois garder à l'esprit que si l'on veut se faire comprendre des autres machines utilisant ce vocabulaire, il ne faut pas trop s'éloigner de sa spécification et les modifications doivent être communiquées au reste du réseau.

Class	Description		Example

Object	URI:	https://www.w3.org/ns/activitystreams#Object	EXAMPLE 1
			{
			"@context": "https://www.w3.org/ns/activitystreams",
			"type": "Object",
			"id": "http://www.test.example/object/1",
			"name": "A Simple, non-specific object"
			}

	Notes:	Describes an object of any kind. The Object type serves as the base type for most of the other kinds of objects defined in the Activity Vocabulary, including other Core types such as Activity, IntransitiveActivity, Collection and OrderedCollection.	
	Disjoint With:	Link	
	Properties:	attachment | attributedTo | audience | content | context | name | endTime | generator | icon | image | inReplyTo | location | preview | published | replies | startTime | summary | tag | updated | url | to | bto | cc | bcc | mediaType | duration	

Figure 3.4 – Définition du terme « *Object* » dans le vocabulaire ActivityStreams

3.2.4 Webfinger

WebFinger est un protocole standardisé par l'IETF (Internet Engineering Task Force). Il permet de trouver des informations au sujet d'un élément [1]ayant un identifiant unique. Le protocole s'inspire de l'e-mail pour définir ces identifiants.

1. L'IETF laisse le terme « élément » à la libre interprétation des personnes voulant implémenter le protocole. Aujourd'hui, ces éléments peuvent être un utilisateur d'une application, un groupement d'utilisateurs, une bibliothèque musicale...

Ce protocole n'est pas explicitement proposé dans la spécification d'ActivityPub [W3C, 2018b] mais il est utilisé par de nombreuses applications fédérées. Il est important car il permet à un utilisateur présent sur une instance d'avoir les informations nécessaires pour découvrir un autre utilisateur présent sur une autre instance avec seulement la connaissance de son identifiant : il est un bon moyen de découvrir ce qui se passe dans le jardin des autres, selon la métaphore des *Walled gardens* [Memetic, 2012]).

Si pour le protocole mail, un identifiant se présente sous la forme `nom@hébergeur.org`, pour le protocole Webfinger, un identifiant est de la forme `acct:nom@instance.org`, *acct* signifiant « account » – le compte d'un utilisateur sur son instance.

3.2.5 HTTP et HTTP Signatures

3.2.5.1 Le protocole

Le protocole `HTTP`, Hypertext Transfer Protocol – littéralement « protocole de transfert hypertexte » – est le protocole le plus utilisé sur Internet depuis 1990. Il permet le transfert de fichiers localisés par une URL entre un client (le navigateur web) et un serveur.

Ces fichiers ou messages peuvent être transférés avec **des en-têtes** décrivant leur type de contenu (format, image, html, son, JSON etc.). Ces en-têtes sont très importantes pour le protocole ActivityPub car elles précisent le type de donnée contenu dans les messages.

On appelle « requêtes (cf. 2.1.4) HTTP » les messages transmis d'un client à un serveur. Dans l'implémentation d'ActivityPub, les serveurs communiquent aussi entre eux par le biais de requêtes HTTP.

3.2.5.2 La sécurité

D'après la spécification d'ActivityPub [W3C, 2018b, chapitre B], toute application implémentant ce protocole doit pouvoir garantir la sécurité de ses utilisateurs :

— L'application doit assurer qu'un personne ne puisse pas voir son identité usurpée par un autre ;
— L'application doit assurer que le message transmis n'a pas pu être modifié par un tiers en cours de route.

Cependant, aucune solution technique n'est proposée par le W3C. La solution retenue *de facto* est **la signature des requêtes HTTP**.

La signature des requêtes HTTP est basée sur la **cryptographie asymétrique**, les deux principes sont expliqués en Annexe D.

Cette notion de cryptographie reste transparente pour l'utilisateur d'une application implémentant ActivityPub qui n'aura jamais à gérer ses clés privées et publiques (figure D.1).

3.3 Développer une preuve de concept

La notion de « preuve de concept », qui vient de l'anglais *proof of concept* est utilisée dans le processus de développement de produits ou d'applications. Souvent incomplète, c'est une réalisation expérimentale concrète qui doit rester simple tout en démontrant la faisabilité d'une idée ou d'une méthode. Cette réalisation peut alors servir de base pour aider au développement de produits similaires : c'est une ressource qui s'ajoute aux plans, documentations ou spécifications existants.

La faisabilité de l'implémentation du protocole ActivityPub a été démontrée notamment par les logiciels Mastodon et PeerTube. Ils prouvent tous les deux que ce standard permet de fédérer plusieurs serveurs proposant les mêmes services mais aussi des services différents : du mico-blogging et de l'hébergement et partage de vidéos. Cependant, ces applications sont loin d'être *simples* : elles proposent de nombreuses fonctionnalités qui vont au-delà de ce qu'une preuve de concept doit présenter. Elles n'ont pas été développées dans le but de devenir une ressource pour les implémentations futures du protocole et leur complexité ne permet pas d'avoir facilement et rapidement les informations sur cette implémentation.

Si le protocole ActivityPub est clairement décrit dans sa spécification [W3C, 2018b], sa mise en place n'est pas évidente et nécessite de consulter et s'approprier le code de ces applications complètes. Il manque alors une application simple, jouant le rôle de preuve de concept, qui n'implémenterait que le minimum nécessaire à la compréhension du protocole.

C'est dans ce cadre que l'objectif de réaliser une application simple implémentant le protocole ActivityPub a été fixé.

3.3.1 Cahier des charges de la preuve de concept

L'objectif est d'implémenter le protocole de fédération *serveur-à-serveur* d'ActivityPub dans une application simple. L'application réalisée doit :

- Gérer plusieurs comptes d'utilisateur·ice·s ;
- Gérer les inscriptions et les connexions ;
- Sécuriser l'inscription et l'authentification ;
- Ne permettre que le minimum d'action aux personnes non inscrites ;

Elle doit permettre à un· utilisateur·ice inscrit·e :

- De poster du contenu textuel ;
- De rechercher des *acteurs* de la fédivers qui ne sont pas inscrites sur l'application et que l'on appelle les *acteurs distants* ;
- D'être recherché par des acteurs distants de la fédivers ;
- De suivre (et arrêter de suivre) les activités d'acteurs choisis, distants ou non, ;
- D'être suivi·e·s (et de ne plus être suivi·e·s) par des acteurs distants ou non ;

Comme précisé dans la spécification du protocole, il faut rappeler que l'application doit s'assurer que :

- Une personne ne puisse pas voir son identité usurpée par une autre ;

— Les message transmis ou reçus n'ont pas pu être modifiés par un tiers en cours de route.

Enfin, il est important que :

— Les outils utilisés pour le développement soient sous licence libre ;
— L'application soit sous licence libre.

La spécification du W3C propose deux modèles d'implémentation : le modèle *client-serveur* et le modèle *serveur-serveur*. La suite de ce travail ne s'intéresse qu'au second modèle. Si des interactions entre client et serveur existent dans l'application, elles ne suivent pas le modèle recommandé par le W3C et ne seront donc pas détaillées par la suite.

3.3.2 Langages de développement

Le choix du langage de programmation d'une application dépend de plusieurs paramètres : expérience, facilité de mise en place etc. De plus, ce travail devant se faire sur un temps limité, il ne fallait pas que la prise en main du langage soit trop longue.

3.3.2.1 Côté client

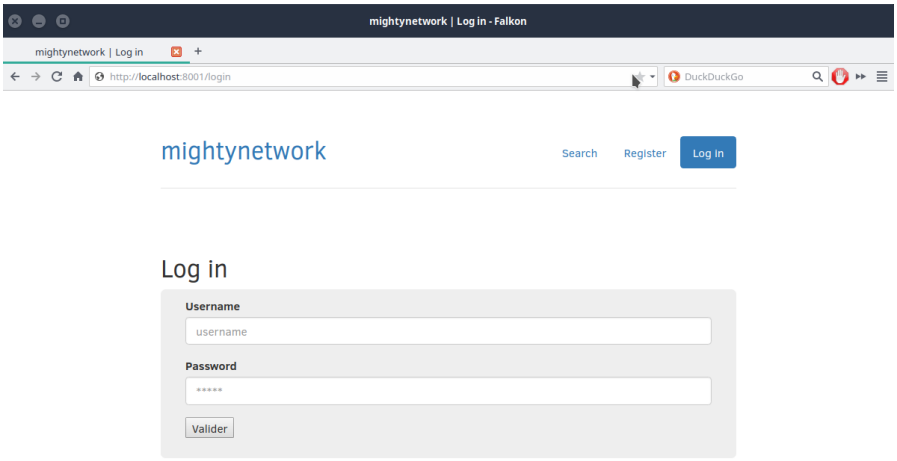


Figure 3.5 – Interface de connexion à l'application réalisée avec un thème Bootstrap

Il a été choisi de développer un client simple pour ce travail car ce n'est pas le cœur de l'implémentation du protocole *serveur-à-serveur* d'ActivityPub. Une interface graphique est toutefois nécessaire pour tester les fonctionnalités de l'application sans avoir à passer par des écritures de requêtes en ligne de commande. Le *framework* Bootstrap fournit

Figure 3.7 – Logo de MongoDB, système de gestion de base de données

librement – et gratuitement – une collection d'outils de création de design de sites web, en langages HTML et CSS, s'installant et se paramétrant facilement. Cela permet d'avoir une interface simple et claire (figure 3.5).

Afin de faire le lien entre les informations données par le serveur et son affichage par le client, le moteur de template *Pug* sera utilisé. Il est souvent celui qui est utilisé avec l'environnement Node.js qui, comme précisé dans la section 3.3.2.2, sera celui utilisé dans ce travail.

L'utilisation de Pug et de Bootstrap permet de minimiser le temps passer au développement de l'application client tout en gardant une interface accessible aux utilisateur·ice·s.

3.3.2.2 Côté serveur

Ce travail est une première expérience de développement côté serveur, il était nécessaire de réduire au maximum la durée de l'apprentissage pour pouvoir atteindre l'objectif fixé. Le langage JavaScript a plusieurs avantages : il a déjà utilisé par l'auteure pour du développement côté client, il est facile à appréhender, de nombreuses documentations existent sur Internet et c'est dans ce langage qu'est développé le logiciel PeerTube dont le développeur est salarié de Framasoft.

Figure 3.6 – Les logos du langage JavaScript, de l'environnement Node.js et du *framework* Express.js

L'environnement Node.js [Parisot, 2018] ainsi que le *framework* Express.js s'imposent rapidement pour une première expérience de développement côté serveur d'autant plus qu'ils sont aussi utilisés pour le développement de PeerTube. Cela permet donc d'utiliser les mêmes modules de l'environnement Node.js que le développeur de PeerTube a utilisé et de s'inspirer de son code pour implémenter ActivityPub.

3.3.3 Gestion de la base de données

L'application à réaliser doit pouvoir gérer une base de données. Cette compétence n'étant pas acquise lors d'expériences précédentes, le choix s'est aussi fait assez naturellement vers le système de gestion de base de données distribué sous license libre MongoDB (3.7) qui appartient à la famille des systèmes dits *NoSQL*. Contrairement à d'autres systèmes de gestion de base de données, MongoDB manipule des objets dans un format très similaire au `JSON` ce qui est très adapté aux contraintes auxquelles ce travail est soumis d'autant plus que c'est le format utilisé par le protocole ActivityPub pour représenter les objets.

3.3.4 Méthode de travail

3.3.4.1 L'environnement de développement intégré

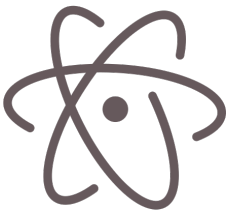


Figure 3.8 – Logo de l'EDI Atom

L'environnement de développement intégré (EDI) utilisé pour développer l'application est l'éditeur de texte libre Atom.

3.3.4.2 La gestion des versions

Figure 3.9 – Logo du gestionnaire de versions Git

Git est un logiciel libre de gestion de versions décentralisé. L'instance utilisée pour publier les différentes versions de ce travail est celle de Framasoft qui se nomme *Framagit*.

Les différentes versions en cours de développement de l'application réalisée se trouvent à cette adresse : `https://framagit.org/narf/mightynetwork`.

3.3.4.3 Le développement

Les phases de documentation, de découverte, et de développement de l'application ont pris une grande partie du temps de ce stage de fin d'étude. Ce développement n'aurait pu

être possible sans l'aide du développeur de PeerTube avec qui un point était fait chaque lundi, d'avril à juillet 2018.

Les grandes étapes de développement sont les suivantes :

— La prise en main de Node.js et de MongoDB ;
— La réalisation d'une application imitant les plateformes de micro-blogging ;
— La sécurisation de cette application ;
— L'implémentation du protocole ActivityPub, expliquée dans la section 3.4. Chaque étape de l'implémentation comprend :
  — La mise en place d'une fonctionnalité ;
  — Le test de cette fonctionnalité avec une installation locale de PeerTube et de Mastodon ;
  — Le débogage jusqu'à validation des tests.
— La réalisation du guide.

Chaque étape de développement s'accompagne d'une étape de documentation sur les modules utilisés, d'implémentation dans l'application et de débogage quand cela est nécessaire.

3.3.4.4 L'application

Des captures d'écran de l'application sont disponible en annexe E. Les sources de l'application sont disponibles sur le dépôt Framagit (section 3.9) ainsi qu'un guide de son installation en local.

L'arborescence de l'application suit les modèles de développement d'une application web côté serveur.

```
/
├── controllers ..... Gestion des requêtes
│   ├── activitypub
│   │   ├── follow.js
│   │   ├── inbox.js
│   │   ├── outbox.js
│   │   └── webfinger.js
│   ├── index.js
│   ├── note.js
│   └── users.js
├── data ..... Base de données
├── helpers ..... Fonctions récurrentes pour chaque objet
│   ├── activitypub
│   │   ├── activity.js
│   │   ├── actor.js
│   │   ├── collection.js
│   │   ├── follow.js
│   │   ├── signature.js
│   │   └── webfinger.js
│   └── note.js
```

```
    ├── users.js
    └── utils.js
├── models ..................... Modèles d'objets reconnus par la base de données
    ├── activitypub
    │   ├── Activity.js
    │   ├── Actor.js
    │   ├── Collection.js
    │   └── Follow.js
    ├── Note.js
    └── User.js
├── node_modules ................................ Sources des modules utilisés
├── public ...................................... Gestion du thème Bootstrap
    ├── css
    │   ├── bootstrap.css
    │   └── style.css
    └── js
        └── bootstrap.js
├── views ....................................... Modèles des pages web en Pug
    ├── error.pug
    ├── follow.pug
    ├── index.pug
    ├── layout.pug
    ├── login.pug
    ├── note.pug
    ├── register.pug
    ├── search.pug
    ├── user.pug
    └── welcome.pug
├── .gitignore
├── apps.js ..................... Fichier principal gérant l'application
├── package-lock.json
├── package.json ..................... Liste des modules utilisés
└── README.md ..................... Guide d'installation en local
```

Les dossiers `controllers`, `helpers`, `models` contiennent tous un sous-dossiers `activitypub`, dans lesquels sont gérés tout ce qui est vraiment spécifique à l'implémentation du protocole. Cependant, cela ne signifie pas que des traces de l'implémentation ne se trouvent pas dans les autres fichiers.

3.4 Guide d'implémentation du protocole ActivityPub

Pour rappel, des définitions nécessaires à la compréhension de ce guide sont données dans le « Glossaire spécifique à ActivityPub ».

3.4.1 Étape 0 : l'application de base

L'application de base dans laquelle le protocole ActivityPub est implémenté doit pouvoir :

— Gérer plusieurs utilisateur·ice·s ;
— Sécuriser leur inscription ainsi que leur connexion ;
— Permettre aux utilisateur·ice·s de l'application de partager des objets divers.

Selon le langage de programmation utilisé, de nombreuses ressources existent sur Internet pour développer un média social simplement.

Par la suite, l'instance de cette application est supposée accessible à l'adresse Web `https://thisinstance.org/`. L'instance sera aussi appelée `thisInstance`.

3.4.1.1 L'utilisateur·ice (`User`)

Dans la base de données MongoDB, un modèle nommé `User` est donc créé, dans lequel est spécifié le format des objets représentant les utilisateur·ice·s de l'application. Le listing 3.5 donne l'exemple d'un objet `User`.

```
{
  "username" : "narf",
  "_id" : "1",
  "password" : "$2a$10$a6ntL6JfdgXkfQHeoof2NuRdgNN3TowEeFdJzCimn9X7O",
  "created_at" : "2018-07-10T11:44:12.349Z"
}
```

Listing 3.5 – Objet `JSON-LD` représentant un utilisateur (`User`) de l'application dans la base de données

Cet objet n'est pas fédéré et chaque _id est unique. Par sécurité, le mot de passe (`"password"`) est chiffré avant d'être stocké dans la base de données.

La sécurisation de leur inscription est gérée par le module `bcrypt` (dans le fichier `/models/User.js, l.36-45`). Celle de leur authentification est gérée par le module `passportjs` (dans le fichier `/controllers/index.js, l.150-198`) ,

3.4.1.2 Les éléments à partager

Ils seront plus détaillés à l'étape 3 (section 3.4.4.1). L'application de base utilisée par la suite aux utilisateur·ice·s d'écrire un message et de le partager aux autres utilisateurs de l'instance. Cela est très similaire au fonctionnement des plateformes de micro-blogging où tous les messages sont publics.

La spécification d'ActivityPub précise que tout objet *fédéré* doit avoir un `type` et un `id` unique. De plus, il est préférable que chaque objet provienne du vocabulaire ActivityStreams (section 3.2.3) bien que d'autres vocabulaires peuvent être spécifiés dans le `"@context"`.
À cette étape, l'application de base est une application centralisée qui ne peut pas communiquer avec d'autres applications. Les étapes suivantes permettent de passer de ce type d'applications à une application de la fédivers.

3.4.2 Étape 1 : l'acteur (`actor`)

3.4.2.1 L'objet `Actor`

La première étape consiste à créer l'objet `Actor` lié à l'objet `User` représentant, cette fois-ci, l'utilisateur·ice dans la fédivers. Cet objet est donc *fédéré* et il est intiment lié à l'objet `User`.

```
1  {
2    "@context": "https://www.w3.org/ns/activitystreams",
3    "type": "Person",
4    "id": "https://thisinstance.org/users/narf",
5    "preferredUsername": "narf",
6    "name": "narf",
7    "following": "https://thisinstance.org/users/narf/following",
8    "followers": "https://thisinstance.org/users/narf/followers",
9    "inbox": "https://thisinstance.org/users/narf/inbox",
10   "outbox": "https://thisinstance.org/users/narf/outbox",
11   "endpoints": {
12     "sharedInbox": "https://thisinstance.org/users/narf/inbox"
13    },
14   "user_id":"1"
15 }
```

Listing 3.6 – Objet `JSON-LD` représentant un acteur (`Actor`) de l'application

Cet objet étant fédéré, il faut lui donner un `type` (listing 3.6, ligne 3) et un `id` (listing 3.6, ligne 4). Dans la spécification d'ActivityPub, si l'instance se trouve à `https://thisinstance.org/`, que le nom de l'`Actor` considéré est `narf`, son `id` peut être :

- `https://thisinstance.org/users/narf` ;
- `https://thisinstance.org/@narf` ;
- `https://thisinstance.org/@/narf` ;
- etc.

Il est préférable que le choix reste cohérent pour tous les objets `Actor` de l'instance bien que l'on puisse choisir d'attribuer un `id` aléatoire à chaque `Actor`. Il est *nécessaire* que l'objet `Actor` présente les clés `inbox` et `outbox` dont le chemin dépend de l'`id`.

Il est préférable que les clés `followers` et `following` soient présentes et leur chemin dépend aussi de l'`id`. Le reste des clés sont optionnels.

L'entrée `"user_id"` n'est pas précisée dans le vocabulaire ActivityStreams mais il permet de faire le lien entre l'`User` et l'`Actor`.

Ainsi, à chaque inscription d'un nouvel utilisateur, un objet `User` est créé, suivi d'un objet `Actor associé` qui seront tous les deux stockés dans la base de données.

La création de ces deux objets se fait dans le fichier `/controllers/index.js`, lignes `56-60`.
L'objet du listing 3.6 est la réponse renvoyée par le serveur à une requête du listing 3.7.

```
1 GET https://thisinstance.org/users/narf HTTP/1.1
2 Accept : application/activity+json
```

Listing 3.7 – Requête ActivityPub `GET` pour récupérer l'objet `acteur` de nom `narf` de l'instance `https://thisinstance.org/`

Si le paramètre `Accept : application/activity+json` n'est pas précisé, ce sera la page HTML de l'`acteur` qui sera affichée. On appelle *requête ActivityPub* les requêtes ayant cet en-tête.

3.4.2.2 Implémenter WebFinger

Réception d'une requête WebFinger
L'implémentation de WebFinger (section 3.2.4) est importante pour qu'une instance distante puisse récupérer les objets `acteurs` de `thisInstance`.

Afin de récupérer l'acteur `narf` de l'instance `thisInstance`, l'instance distante va effectuer une requête HTTP auprès de `thisInstance`. La requête reçue est celle du Listing 3.8.

```
1 GET /.well-known/webfinger?resource=narf@thisinstance.org HTTP/1.1
```

Listing 3.8 – Requête `GET` WebFinger reçue

Cette requête – qui n'est pas une requête ActivityPub – signifie littéralement « Je souhaite récupérer les informations WebFinger de l'`acteur` dont le nom est `narf` et qui est sur l'instance `thisInstance` ». L'application doit répondre à cette requête en exposant l'objet `WebFinger`(listing 3.9).

Cet objet précise dont l'`href` de l'`acteur` i.e. l'adresse à laquelle faire une requête pour récupérer l'objet représentant cet `acteur` et le `type` de contenu i.e. le type d'`Accept` à préciser pour récupérer l'objet voulu (listing 3.10).

Cette implémentation se fait dans le fichier `controllers/activitypub/webfinger.js`.

```
{
  "subject": "acct:narf@thisinstance.org",
  "aliases": [
    "https://thisinstance.org/users/narf"
  ],
  "links": [{
      "rel": "self",
      "type": "application/activity+json",
      "href": "https://thisinstance.org/users/narf"
    }
  ]
}
```

Listing 3.9 – Réponse du serveur à une requête `GET` Webfinger

Recherche d'un `acteur` distant

Il est aussi nécessaire qu'un `acteur` de `thisInstance` puisse rechercher un acteur distant de la fédivers. Une requête similaire est à celle du listing 3.8 doit être réalisée par le serveur vers un serveur distant. Il est toutefois nécessaire que l'`acteur` de `thisInstance` connaisse l'identifiant WebFinger de l'acteur distant afin de ne pas recevoir une erreur en réponse à la requête. Pour connaitre l'identifiant WebFinger, il suffit de connaître le `preferredUsername` ainsi que l'instance de l'`actor`.

```
{
   "type": "application/activity+json",
   "href": "https://thisinstance.org/users/narf"
}
```

Listing 3.10 – Informations intéressantes du Listing 3.9

À la requête WebFinger, `thisInstance` reçoit la réponse (listing 3.9) de laquelle est elle retire les informations importantes (listing 3.10).

Enfin, pour récupérer l'objet `acteur` (listing 3.6) de la recherche, il est nécessaire d'effectuer la requête du listing 3.7 avec l'adresse donnée par `href`.

3.4.3 Étape 2 : la boîte de réception (`inbox`)

L'`inbox` d'un `acteur` est le chemin où il reçoit toutes les requête ActivityPub dont il est destinataire (listing 3.6, ligne 9). Dans cette application, c'est aussi l'élément qui gère ce qui est fait des objets ou activités reçues.

Son implémentation se fait dans le fichier `/controllers/activitypub/inbox.js`.

L'`inbox` ne gère que les requêtes ActivityPub `POST` qui sont faites aux `/inbox` des acteurs de l'instance. Elle récupère les informations données par cette requête qui doivent

être de la forme d'activité (section 3.4.5) et les traite en fonction du `type` de ces activités comme l'extrait **??** le montre.

```
var express = require('express');
var router = express.Router();
// (...)
router.post('/', function (req, res) {
  var activity = req.body; // Activité reçue

  if (activity.type === 'Create') {
    // Fonction qui définit ce qui est fait d'une activité de type 'Create'
  }

  if (activity.type === 'Follow') {
    // Fonction qui définit ce qui est fait d'une activité de type 'Follow'
  }

  if (activity.type === 'Accept') {
    // Fonction qui définit ce qui est fait d'une activité de type 'Accept'
  }

  if (activity.type === 'Undo') {
      // Fonction qui définit ce qui est fait d'une activité de type 'Undo'
  }
});
module.exports = router;
```

Listing 3.11 – Extrait du fichier `/controllers/activitypub/inbox.js`

3.4.4 Étape 3 : les objets (`object`)

3.4.4.1 Un objet

Les différents type d'`objets` qui peuvent être utilisés dans la fédivers sont défini dans le vocabulaire ActivityStreams. Dans les applications existantes de la fédivers, les objets qui sont utilisés sont :

- `Article` par Plume [Gelez, 2018], un moteur de blog ;
- `Audio` par FunkWhale [Berriot, 2018, Rf, 2018], une application d'hébergement et de diffusion de musique ;
- `Event` par une application permettant d'organiser des évènements, en cours de développement ;
- `Image` par PixelFed [dansup, 2018], une application de partage d'images
- `Note` majoritairement par Mastodon [Gargon, 2016] ou Pleroma [Pleroma, 2017], applications de micro-blogging mais aussi par les autres applications qui l'utilisent

pour les commentaires.
— `Video` par PeerTube [Chocobozzz, 2018a, Hermann, 2018], application d'hébergement et de diffusion de vidéos

La compréhension de tous les objets ne doit pas nécessairement être implémentée pour pouvoir faire partie de la fédivers. L'application développée ici n'implémente que les objets de type `Note`.

3.4.4.2 Une `Note`

C'est l'objet le plus implémenté dans la fédivers, ce qui en fait l'objet le plus facile à implémenter car les tests peuvent se faire avec toutes les applications de la fédération.

Dans l'application réalisée, il existe un champ que l'utilisateur·ice peut remplir pour écrire un message à partager au reste de la fédivers (figure (c) de E). Lorsque l'utilisateur·ice appuie sur le bouton `Envoyer`, une requête est envoyée au serveur avec le contenu du message (listing 3.12).

```
1 POST / HTTP/1.1
2 Content-Type : application/json
3  {
4     "clés" : "valeurs",
5     "content" : "Hello world !"
6  }
```

Listing 3.12 – Extrait de la requête reçue par le serveur quand un·e utilisateur·ice poste un message

Cette action est typiquement du type *client-serveur*, son implémentation n'est pas précisée dans ce guide qui n'aborde que de partie *serveur-serveur*. La récupération de ce message et sa transformation en objet `Note` se fait dans le fichier `controllers/note.js`. Ainsi, si le serveur reçoit une requête du type listing 3.12, il faut récupérer le `content` et créer l'objet `Note` (listing 3.13).

Comme tout autre objet, un objet de `"type" : "Note"` doit avoir un `id` unique. Lorsqu'un élément est enregistré dans une base de données MongoDB, il lui est toujours attribué un `_id` unique : dans l'application, cette `_id` est récupéré pour construire l'`id` unique de la note sous la forme d'une URL (listing 3.13, ligne 4).

La clé `"to"` référence les destinataires du message. Dans ce cas, elle représente une liste :
— `"https://www.w3.org/ns/activitystreams#Public"` signifie, en vocabulaire ActivityStreams, que le message est public i.e. visible par toute la fédivers.
— `"https://thisinstance.org/users/narf/followers"` signifie que ce message est destiné aux `followers` de l'utilisatrice "narf".

La clé `"attributedTo"` désigne l'`acteur` émettant le message, identifié par son `id`.

La clé `"content"` reprend le message écrit du listing 3.12. La clé `"published"` date le message. Cette datation est obligatoire pour des questions de sécurité, comme nous le

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "id" : "https://thisinstance.org/users/narf/note/5b44c2974a67a67eec472d82",
  "to" : [ "https://www.w3.org/ns/activitystreams#Public" ],
  "cc" : [ "https://thisinstance.org/users/narf/followers" ],
  "attributedTo":"https://thisinstance.org/users/narf",
  "content": "Hello world !",
  "published":"2018-07-10T14:28:39.889Z",
}
```

Listing 3.13 – Objet `JSON-LD` représentant une `Note`

verrons en section 3.4.6.

Comme cela sera précisé dans la section 3.4.5.1, bien qu'il suive un modèle d'objet ActivityStreams, l'objet `Note` n'est que le message et il n'est pas fédéré. Dans l'application, il a été choisi de le garder en base de données.

3.4.5 Étape 4 : les activités (`activities`)

Ce sont des objets qui représentent les actions effectuées par un `actor`. Les entrées dépendent du type d'activité mais les deux principales sont l'`actor` et l'`object`. Tout objet ActivityStreams peut être objet d'une activité. Dans l'application étudiée, les types d'activités implémentés sont :

- `Create`
- `Follow`
- `Accept`
- `Undo`

Les `activities` sont, comme les `actors`, des objets qui sont fédérés. Ainsi, lorsqu'une activité est réalisée, une requête ActivityPub est faite aux destinataires de cette activité (listing 3.14).

Cette requête suit la méthode `'POST'`, elle contient l'objet `JSON activityObject` qui sera défini par la suite. C'est une requête ActivityPub donc elle doit préciser l'en-tête `'Accept':'application/activity+json'` (listing 3.14, lignes 6-8) pour que le serveur récepteur puisse la traiter comme un objet ActivityPub. Elle doit être adressée à l'`inbox` du destinataire (section 3.11).

3.4.5.1 L'activité `Create`

L'activité `Create` est l'enveloppe dans laquelle se trouve l'objet `Note` (listing 3.13), qui est le message qui n'est pas fédéré.

L'objet `Note` est alors comparable à une lettre : on y retrouve un en-tête avec l'expé-

```
1  var requestOptions = {
2    url: "https://anotherinstance.org/users/narf_follower/inbox",
3    json: true,
4    method: 'POST',
5    body: activityObject,
6    headers: {
7      'Accept': 'application/activity+json'
8    }
9  };
10
11 request(requestOption);
```

Listing 3.14 – Requête ActivityPub - Envoi d'une activité à l'acteur `narf_follower`

diteur et le destinataire, une date, le contenu du message. L'activité `Create` représente l'enveloppe dans laquelle la lettre est postée. La lettre est alors l'objet de l'enveloppe.

Comme tout autre objet, l'activité `Create` (listing 3.15) doit avoir un `id` unique. Comme cet objet est intement lié à la `Note`, son `id` est celui de la `Note` auquel on concatène `/activity` (listing 3.15, ligne 3).

```
1  {
2   "@context" : "https://www.w3.org/ns/activitystreams",
3    "id" : "https://thisinstance.org/users/narf/note/5b44c2974a67a67eec472d82/activity",
4    "type" : "Create",
5    "actor" : "https://thisinstance.org/users/narf",
6    "object":
7     {
8      "type": "Note",
9      "id" : "https://thisinstance.org/users/narf/note/5b44c2974a67a67eec472d82",
10         "to" : [ "https://www.w3.org/ns/activitystreams#Public" ],
11     "cc" : [ "https://thisinstance.org/users/narf/followers" ],
12     "attributedTo":"https://thisinstance.org/users/narf",
13     "content": "Hello world !",
14     "published":"2018-07-10T14:28:39.889Z",
15      },
16   "published" : "2018-07-10T14:28:39.889Z" }
```

Listing 3.15 – Objet `JSON-LD` représentant une activité `Create`

On retrouve bien la `Note` comme objet de l'activité. Il est important de remarquer que la clé `"published"` de l'activité a la même valeur que celle de la `Note` : l'objet `Create` est créé en même temps que la note, dans la base de données. Cette entrée est très importante pour la sécurité (section 3.4.6).

Le lien `https://thisinstance.org/users/narf/followers` renvoie à une liste (section 3.4.7) des `id` des tous les acteurs qui *suivent* l'acteur `narf`. Cette activité sera ensuite l'objet d'une requête `HTTP POST` vers le lien `inbox` (section 3.4.3) de tous les `acteurs` appartenant à la liste des `followers`.

Toute cette implémentation se trouve dans le fichier `/controllers/note.js (l.53-99)`

3.4.5.2 Les activités `Follow` et `Accept`

Les activités `Follow` et `Accept` vont de paire.

La première représente l'action de *suivre* un `acteur`. Si l'acteur `narf` *suit* l'acteur `ren`, alors `narf` fait partie des *followers* de `ren` et recevra ainsi toutes les Notes dont une des valeurs de la clé `"cc"` est `https://thisinstance.org/users/ren/followers`.

Si un·e utilisateur·ice souhaite suivre – faire une *follow request* à – une autre personne, il suffit de cliquer sur la bouton `Follow` présent sur la page de cette personne (figure E, (a) et (b)).

Cette action est à nouveau du type *client-serveur* que l'on ne traite pas dans ce guide.

Au niveau du serveur, cette action génère l'activié `Follow` (listing 3.16).

```
1 {
2   "@context": "https://www.w3.org/ns/activitystreams",
3   "id" : "https://thisinstance.org/users/narf/follows/5b449c172488de71310185d5",
4   "type" : "Follow",
5   "summary" : "narf sends a follow request to ren",
6   "actor" : "https://thisinstance.org/users/narf",
7   "object" : "https://anotherinstance.org/users/ren",
8 }
```

Listing 3.16 – Objet `JSON-LD` représentant une activité `Follow`

Elle est équivalente à une demande d'ajout à la liste `https://anotherinstance.org/users/ren/followers`. Cette activité est envoyée à l'`inbox` de l'acteur `ren`, avec la requête du listing 3.14. Cette activité, comme tout autre objet, a un `id` unique (listing 3.16, ligne 3).

Dans l'application, le choix a été fait d'accepter automatiquement toutes les demandes de ce type. Ainsi lorsqu'une requête ayant pour objet un activité de type `Follow` est reçue dans l'`inbox`, une requête ayant pour objet une activité de type `Accept` (listing 3.17) est renvoyée.

L'`object` d'une activité de type `Accept` est l'activité à laquelle elle répond : dans ce cas c'est l'activité de type `Follow` reçue auparavant.

En renvoyant cette activité à `narf`,

— l'acteur `ren` voit sa liste de `followers` (`https://anotherinstance.org/users/ren/followers`) mise à jour ;

— l'acteur `narf` voit sa liste de `following` (`https://thisinstance.org/users/narf/following`) mise à jour

```
1  {
2   "@context" : "https://www.w3.org/ns/activitystreams",
3   "id" : "https://anotherinstance.org/users/ren/accept/5b449c172488de71310185d5",
4   "type" : "Accept",
5   "actor" : "https://anotherinstance.org/users/ren",
6   "object" :
7    {
8     "id" : "https://thisinstance.org/users/narf/follows/5b449c172488de71310185d5",
9     "type" : "Follow",
10    "summary" : "narf sends a follow request to ren",
11    "actor" : "https://thisinstance.org/users/narf",
12    "object" : "https://anotherinstance.org/users/ren",
13   }
14 }
```

Listing 3.17 – Objet `JSON-LD` représentant une activité `Accept`

Cette activité à une `id` unique qui reprend l'`id` de l'activité `Follow` à laquelle elle répond, en changeant « `follows` » par « `accept` » (listing 3.17, ligne 3).

L'activité de type `Reject` n'est pas implémentée dans cette application. Elle permet cependant de *rejeter* ou *refuser* une activité de type `Follow`. Elle se construit de la même manière que l'activité de type `Accept`.

3.4.5.3 L'activité `Undo`

L'activité de type `Undo` est utilisée dans l'application pour reproduire l'action d'« arrêter de *suivre* un `acteur` ». Elle permet d'annuler – en anglais *undo* – une précédente activité de type `Follow` même si celle-ci a été acceptée.

Elle a donc pour objet l'activité de type `Follow` qu'elle souhaite annuler (listing 3.18).

Cette activité à une `id` unique qui reprend l'`id` de l'activité `Follow` qu'elle souhaite annuler, en changeant « `follows` » par « `unfollow` » (listing 3.18, ligne 3).

3.4.6 Étape 5 : la sécurité

En réalité, toutes les étapes présentées précédemment ne sont pas suffisantes pour que l'application puisse intéragir avec le reste du fédiverse. Des règles de sécurité doivent être respectées afin que des instances de Mastodon ou de PeerTube acceptent les messages venant de l'application. Pour rappel, les voici :

1. Les message transmis ou reçus ne peuvent pas être modifiés par un tiers en cours de route.
2. Une personne ne doit pas voir son identité usurpée par une autre ;

Les solutions techniques pour suivre ses règles ne sont pas précisées par la spécification

```
1  {
2    "@context": "https://www.w3.org/ns/activitystreams",
3    "id" : "https://thisinstance.org/users/narf/unfollow/5b449c172488de71310185d5",
4    "type" : "Undo",
5    "summary" : "narf unfollows ren",
6    "actor" : "https://thisinstance.org/users/narf",
7    "object" :
8     {
9      "id" : "https://thisinstance.org/users/narf/follows/5b449c172488de71310185d5",
10     "type" : "Follow",
11     "summary" : "narf sends a follow request to ren",
12     "actor" : "https://thisinstance.org/users/narf",
13     "object" : "https://anotherinstance.org/users/ren",
14    }
15 }
```

Listing 3.18 – Objet `JSON-LD` représentant une activité `Undo`

du protocole ActivityPub. L'application s'inspire donc des solutions mises en place par Mastodon.

3.4.6.1 Cryptographie asymétrique

Afin de répondre à ces deux contraintes, il est tout d'abord nécessaire d'introduire la méthode de cryptographie asymétrique dans l'application. Cette méthode est expliquée en annexe D.

Tous les `acteurs` de l'application doivent avoir une clé privée et une clé publique. Lors de l'inscription d'un·e utilisateur·ice, le module `pem` [npm, 2018] est utilisé pour lui générer une clé privée. La clé publique est déduite de cette clé privée. Cela se fait dans la fonction `createLocalActor`, qui se trouve dans `models/activitypub/Actor.js`.

Une personne ayant accès à la base de données peut donc voir la représentation de l'acteur comme sur le listing 3.19.

Bien évidemment, la clé privée – `privateKey` – n'est jamais exposée publiquement. Ainsi lorsque l'`id https://thisinstance.org/users/narf` fait l'objet d'une requête ActivityPub i.e. avec l'en-tête `'Accept':'application/activity+json`, la réponse est l'objet `actor` du listing 3.19 sans la clé `privateKey`.

Il est aussi important de remarquer que la valeur associée à la `"@context"` a changé. Le vocabulaire `https://w3id.org/security/v1` est nécessaire pour comprendre les entrées `publicKey` et `privateKey`. Il faut, de plus, préciser l'algorithme de chiffrement utilisé : ici `RsaSignature2017` (listing 3.19, lignes 4, 6).

3.4.6.2 Signature des activités

Cette technique permet de répondre à la première contrainte citée précédemment. Elle se base sur la méthode de signature et vérification décrite en annexe D.2.

Lorsqu'une activité est signée avec la clé privée de l'`acteur` qui la crée, si l'activité est interceptée par une personne tierce, cette dernière peut potentiellement la modifier mais ne pourra pas la signer. L'activité ré-émise ne sera pas vérifiée. Ainsi, à chaque fois qu'une activité est créée, elle doit être signée par l'émetteur avant d'être envoyée dans une requête ActivityPub (listing 3.20).

Le module `jsonld-signature` [Chocobozzz, 2018b] est utilisé pour effectuer cette signature et son implémentation se fait dans la fonction `signObject`, dans `/helpers/acivitypub/signature.js`.

3.4.6.3 Signature des requêtes ActivityPub

Cette technique permet de répondre à la deuxième contrainte citée précédemment. Elle se base sur la méthode *HTTP Signature* est expliquée en annexe D.3. Elle est implémentée dans la fonction `postSignedObject` qui se trouve dans `/helpers/activitypub/signature.js`.

Le listing D.1 résume l'implémentation :

— Le `keyId` est l'identifiant WebFinger de l'acteur : par exemple `acct:narf@thisinstance.org` ou encore `ren@anotherinstance.org` ;

— `"body"` (ligne 13) contient l'activitée signée construite précédemment.

```
{
  "@context": [
     "https://www.w3.org/ns/activitystreams",
     "https://w3id.org/security/v1",
     {
      "RsaSignature2017": "https://w3id.org/security#RsaSignature2017"
     }
   ],
  "id": "http://thisinstance.org/users/narf",
  "type": "Person",
  "following": "http://thisinstance.org/users/narf/following",
  "followers": "http://thisinstance.org/users/narf/followers",
  "inbox": "http://thisinstance.org/users/narf/inbox",
  "outbox": "http://thisinstance.org/users/narf/outbox",
  "preferredUsername": "narf",
  "name": "narf",
  "url": "http://thisinstance.org/users/narf",
  "endpoints": {
     "sharedInbox": "http://thisinstance.org/users/narf/inbox"
    },
  "publicKey": {
     "owner": "http://thisinstance.org/users/narf",
     "id": "http://thisinstance.org/users/narf#main-key",
     "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0B
     AQEFAAOCAQ8AMIIBCgKCAQEAwzxINWRxQESi2sDqdDB0\nHl+sm0reSiHjhJJUeu9M
     d3m5t7/+Morljb7KtReMWsVhzYHHJrlCnYiMJv8jC9Sz\nZmBUpp6LO8HewdlQQBOF
     tf27uBLNqjxjiSeiffZMT9NbuCmJNesXHKQXy8a56w59\nycEn+b8Hf0WYYYdJRpwr
     BCEb8YYGoHVefz1/mg4ADMngBqfIkiF1Sgvnf1ql6VYz\n81T1YTlqBoRGjwVLGJa
     OunoRTwVsEkTDnxVQRaOSONbVfmaqBzywCcHdt+m22xHf\nufaCQ0RD2cb7YQ/8OY
     yfHEcqIGSKjQ04gdA/pnK1ryKSdhhHNWDkeWKVMbmx822
     Q\nkQIDAQAB\n-----END PUBLIC KEY-----"
    },
  "privateKey": "-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAwzxIN
  WRxQESi2sDqdDBOHl+sm0reSiHjhJJUeu9Md3m5t7/+\nMorljb7KtReMWsVhzYHHJrlC
  nYiMJv8jC9SzZmBUpp6LO8HewdlQQBOFtf27uBLN\nqjxjiSeiffZMT9NbuCmJNesXHKQ
  Xy8a56w59ycEn+b8Hf0WYYYdJRpwrBCEb8YYG\noHVefz1/mg4ADMngBqfIkiF1Sgvnf1
  ql6VYz81T1YTlqBoRGjwVLGJaOunoRTwVs\nEkTDnxVQRaOSONbVfmaqBzywCcHdt+m22
  xHfufaCQ0RD2cb7YQ/8OYyfHEcqIGSK\njQ04gdA/pnK1ryKSdhhHNWDkeWKVMbmx822Qk
  QIDAQABAoIBABa3B8u2gPqyQaRj\n96NHE+uIxCNZRZ2obbk58TxQZTwtXG9FlkS5JlYk
  kZ8aOOlAV826QLp/qYF8ppnO\n63vv5mON56tvHNH6D3nMV+rsmoz/FXOoKueYmRgXFW/
  EXUKYW76EjqNIU+sQKHic\nCXuQtw/nOkhzcmLbbQxD2kGt389d+r0Zj+4qxnobfJVzE3
  yvjJezpqpjYvPLQzCU\nJo3hp+gFXFeV6OKeK/wiI8gj7CT4iT884DRDf1KzWPnosFp1
  yql39TFOCkvue98+\net3jv2JFQ1qNZJIXDMuH/zphhyjmwuyKjiK6+CMz6+eT3IzUw3
  qFYShg/EVkOks/\n7fS7mXECgYEA57RGzyk\n-----END RSA PRIVATE KEY-----"
}
```

Listing 3.19 – Objet représentant l' `Actor` au complet dans la base de donnée. L'objet ActivityPub public n'expose pas l'entrée `privateKey`.

```
{
    "@context":
      [ "https://www.w3.org/ns/activitystreams",
        "https://w3id.org/security/v1",
            {
             "RsaSignature2017": "https://w3id.org/security#RsaSignature2017"
            }
      ],
    "id": "https://thisinstance.org/users/narf/follows/5b449c172488de71310185d5",
    "type": "Follow",
    "summary": "narf sends a follow request to ren",
    "actor": "https://thisinstance.org/users/narf",
    "object": "https://anotherinstance.org/users/ren",
    "signature":
    { "type": "RsaSignature2017",
      "created": "2018-09-05T08:49:37Z",
      "creator": "https://thisinstance.org/users/narf",
      "signatureValue": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19
      ..ok66fypV2W4MLmuElzshes_RTOuYvJuWEN1F_ZQOrCtHDOY1RE_CaAicc7JEsqJlApRO899mg
      ufjzmMlWQQKLIrFpnhfVE-XMdMnw-GefDADXMpOOVDZmmbks9LkwL5Efjm5xoXBctkTOvsyuoAi
      93H6ev_OV-Vcp2EJhjlu27DIjoIAHt7u7OxjXr4mnROutmEW_tNnXEtMCpPKeIBHrS-VopNCUQp
      R3eGDkGIsElEm95VPk6D7lV8pnJn60IsaEMRvI1vqj0Z8m4Cks-kpdONK6Skv0OAr5awHfyLOts
      OnIs-gOkOngUtpM9InTNIsoMPXHiBo1j02hePkHZL7Pg"
    }
}
```

Listing 3.20 – Activity `Follow` signée

3.4.7 Étape 6 : les listes (`collections`)

Les objets `collections` sont des listes d'autres objets : activités, acteurs etc. Leurs implémentations ne sont pas nécessaires au bon fonctionnement de l'application dans la fédivers. De ce fait, elle ne sera pas très détaillée.

Les identifiants suivant exposent des `collections` lorsqu'ils reçoivent une requête GET ActiityPub de la forme du listing 3.7 :

— https ://thisinstance.org/users/narf/outbox
— https ://thisinstance.org/users/narf/followers
— https ://thisinstance.org/users/narf/following

Le listing 3.21 est un exemple de `collection` dite *ordonnée*.

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://w3id.org/security/v1"
  ],
  "id": "https://thisinstance.org/users/narf/followers",
  "type": "OrderedCollection",
  "totalItems": 3,
  "orderedItems": [
    "https://anotherinstance.org/users/ren",
    "https://yetanotherinstance.social/users/pierre",
    "https://thisinstance.org/users/nathalie"
  ]
}
```

Listing 3.21 – Collection ordonnée des `followers` de `narf`

Conclusion générale

Le travail présenté avait pour objectif de vulgariser les concepts et les techniques de la fédération des médias sociaux numériques. Deux types de publics (non nécessairement distincts) étaient visés : un public non-initié aux enjeux de la fédération et un public de développeurs et développeuses recherchant de la documentation pour implémenter ce protocole dans leurs applications.

Afin de réaliser ce travail, il a tout d'abord fallu comprendre les enjeux et les principes de la fédération. Un retour à la base de l'architecture des médias sociaux était nécessaire et a été présentée pour rendre le travail de vulgarisation plus complet. Une grande partie des documents expliquant les protocoles de fédération est en anglais, il était nécessaire de les rendre accessibles aux non-anglophones. Pour cela, un glossaire franaçais a été réalisé ainsi qu'une description non technique des enjeux. Il a ensuite fallu prendre en main le protocole ActivityPub afin de réaliser une preuve de concept documentée. Cette prise en main a nécessité l'étude de très nombreux autres protocoles sur lesquels se base ActivityPub. La preuve de concept réalisée ainsi que sa documentation sont de réels apports aux outils de prise en main du protocole car aucune application *simple* n'avait été réalisée avant l'implémentation d'ActivityPub dans les applications complexes que sont PeerTube ou Mastodon.

Les différents documents textuels réalisés lors de ce travail seront diffusés par la suite sur le blog de l'association Framasoft, sous la license CC-BY-SA, et la preuve de concept sur la forge logiciel de Framasoft, sous la license AGPL, pour qu'ils puissent être utilisés, étudiés et améliorés librement.

Par ailleurs, un travail de philosophie a été réalisé en parallèle du présent travail technique. Il existe de nombreuses raisons qui peuvent pousser à la recherche de technologies alternatives : avoir plus de diversité, ne pas dépendre d'un seul acteur, favoriser la concurrence, donner plus de choix aux internautes, innover, effectuer des prouesses techniques... Comme présenté dans le contexte, ce sont plutôt des raisons politiques qui ont motivé dans ce présent travail. L'association Framasoft propose de réfléchir aux enjeux d'Internet en dehors du cadre marchand qui est prédominant chez les entreprises développant les outils numériques massivement utilisés et dans la visée d'une société de contribution. Les médias sociaux font aujourd'hui partie du quotidien des individus, ils favorisent les échanges d'idées, de connaissances, de culture, ils permettent de s'organiser, de collaborer, de faire ensemble. Alors que les géants du Web se posent comme inévitables lorsque l'on pense les médias sociaux, le travail de philosophie cherche a montrer qu'il

est de la responsabilité de chacun, à son échelle, et de la société d'offrir la possibilité de faire sans eux. Construire ce type d'alternatives, dont le modèle ne repose pas sur la marchandisation mais sur la contributions, est nécessaire pour renouer avec des valeurs de liberté et de solidarité qui ont donné naissance à l'Internet.

Références

Berners-Lee, T. ; Hendler, J. L. O. (2001). The Semantic Web. *Scientific American.*

Berriot, E. (2018). Funkwhale, a self-hosted tribute to Grooveshark.com. *https://code.eliotberriot.com/funkwhale/funkwhale*.

Bortzmeyer, S. (2015). Centralisé, décentralisé, pair à pair, quels mots pour l'architecture des systèmes répartis ? *JRES.*

Chocobozzz (2018a). Federated (ActivityPub) video streaming platform using P2P (BitTorrent) directly in the web browser with WebTorrent. *https://framagit.org/chocobozzz/PeerTube*.

Chocobozzz (2018b). JSON-LD Signatures with Rsa2017. *https://github.com/Chocobozzz/jsonld-signatures#rsa2017*.

Dachary, L. (2009). Les Régimes de l'open source : solidarité, innovation et modèles d'affaires. *Centre de gestion scientifique, Mines ParisTech.*

dansup (2018). PixelFed is a federated social image sharing platform, similar to Instagram. *https://github.com/pixelfed/pixelfed*.

Framasoft (2014). Dégooglisons Internet. *https://degooglisons-internet.org/*.

Framasoft (2017a). Contributopia : dégoogliser ne suffit pas ! *https://framablog.org/2017/10/09/contributopia-degoogliser-ne-suffit-pas/*.

Framasoft (2017b). Site internet du projet Contributopia. *https://contributopia.org/fr/*.

Gargon (2016). Mastodon is a free, open-source social network server based on open web protocols. *https://github.com/tootsuite/mastodon*.

Gelez, B. (2018). Plume, a federated blogging engine. *https://github.com/Plume-org/Plume*.

GNU (1996). Qu'est ce que le logiciel libre ? *https://www.gnu.org/philosophy/free-sw.fr.html*.

Goffi (2015). Centralisé, décentralisé, P2P, mais c'est quoi tout ça ? *goffi.org*.

Gosset, P.-Y. (2018). Conférence « Contributopia : Dégoogliser ne suffit pas ! ». *Rencontre Mondiales du Logiciel Libre.http://www.canalc2.tv/video/15197*.

Guilloux, M. (2018). Quelles sont les entreprises qui contribuent le plus aux projets open source ? *Developpez.com.*

Hermann, V. (2018). PeerTube : le « YouTube décentralisé » passe en bêta publique. *Next Inpact*.

IETF. HTTP Signatures. *https://tools.ietf.org/id/draft-cavage-http-signatures-01.html*.

JoinPeerTube (2018). Les instances de PeerTube. *https://instances.joinpeertube.org/instances*.

Jourdane, N. (2018). Comment réparer les médias sociaux (et faire encore mieux). *Framablog*.

Memetic, D. (2012). Escaping the Walled Gardens in the Cloud. *Tech-FAQ.com*.

MNM (2018). Mastodon Network Overview. *https://dashboards.mnm.social/*.

npm (2018). Create private keys and certificates with node.js. *https://www.npmjs.com/package/pem*.

Parisot, T. (2018). Node.js – Apprendre par la pratique. *https://oncletom.io/node.js/#sommaire*.

Pleroma (2017). Pleroma is an OStatus-compatible social networking server written in Elixir, compatible with GNU Social and Mastodon. *https://git.pleroma.social/pleroma/pleroma*.

PwC (2017). Global Top 100 Companies by market capitalisation.

Rf, N. (2018). Funkwhale, les baleines mélomanes libres et décentralisées. *Framablog*.

Stallman, R. (2007). En quoi l'open source perd de vue l'éthique du logiciel libre. *https://www.gnu.org/philosophy/open-source-misses-the-point.fr.html*.

W3C (2011). ActivityStreams 2.0 Terms. *https://www.w3.org/ns/activitystreams*.

W3C (2014). W3C launches push for Social Web Application interoperabilité. *https://www.w3.org/blog/news/archives/3958*.

W3C (2018a). ActivityPub is now a W3C recommendation. *W3C News*.

W3C (2018b). Spécification ActivityPub - W3C Recommendation. *https://www.w3.org/TR/activitypub/*.

Annexe A

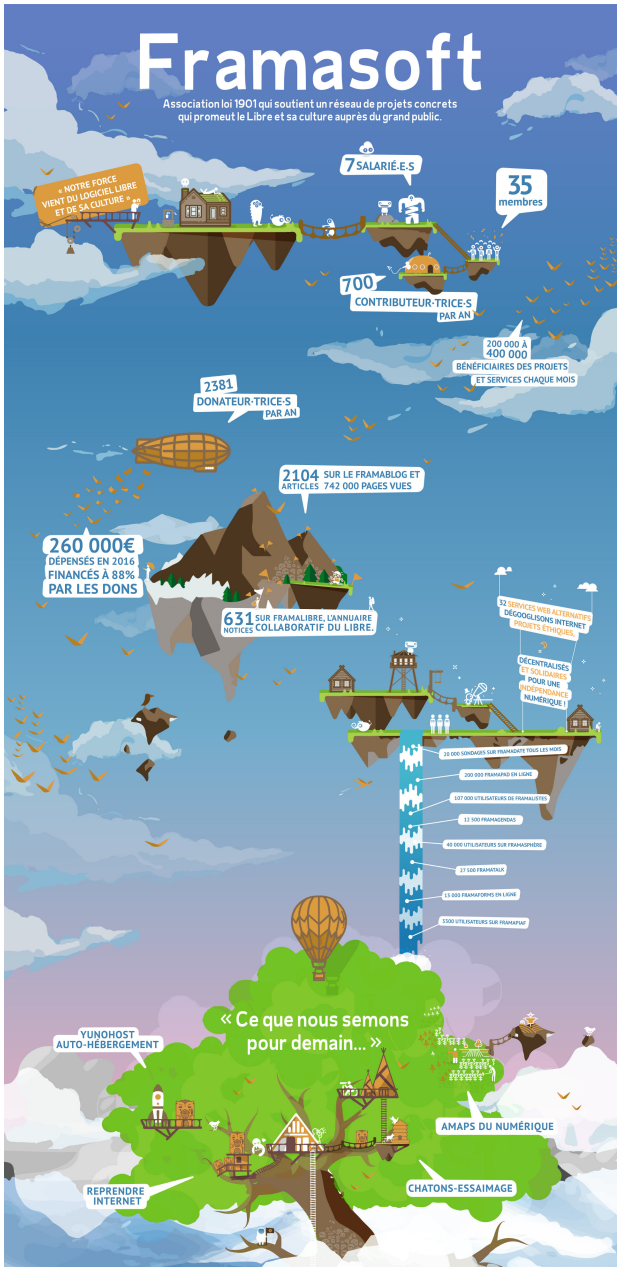


FIGURE A.1 – Infographie de présentation de l'association Framasoft

Annexe B

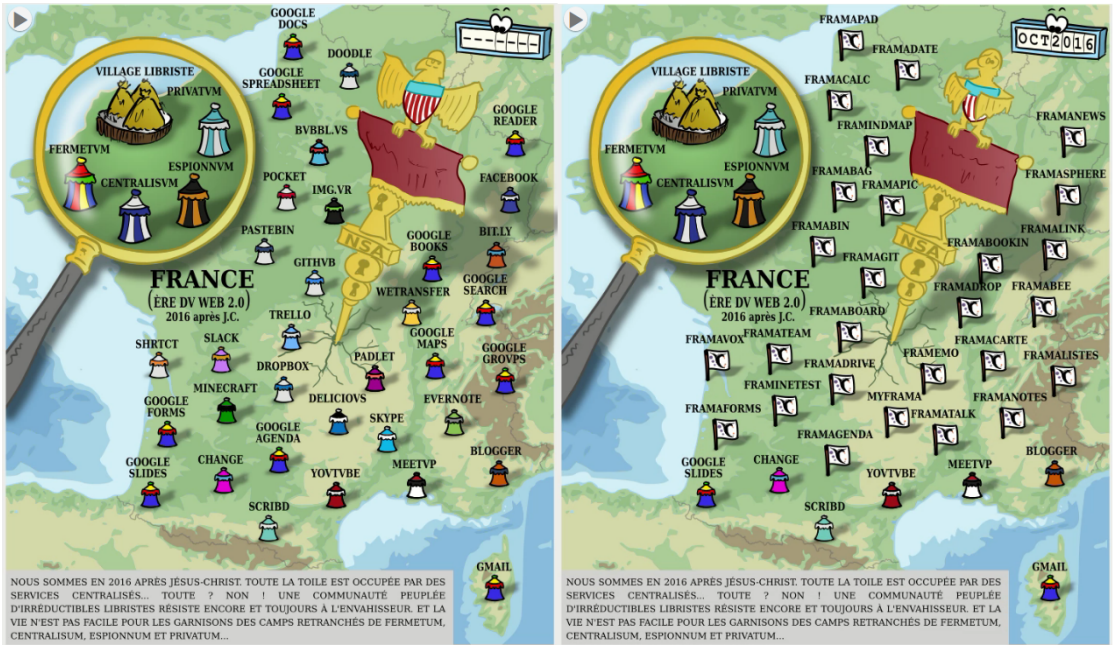


FIGURE B.1 – À gauche, une carte des services centralisés - À droite, la carte des services alternatifs proposés par Framasoft en Octobre 2016

Annexe C

PeerTube : une alternative décentralisée à YouTube

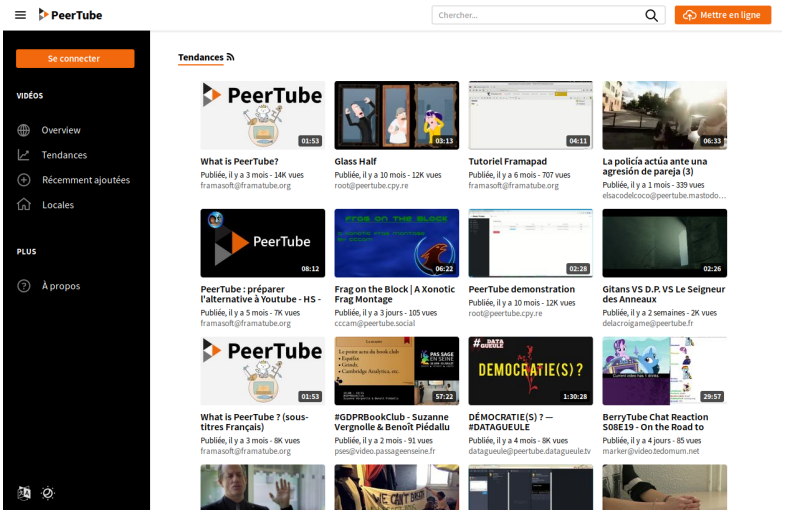


FIGURE C.1 – Capture de l'interface de PeerTube

PeerTube est un logiciel libre qui permet de créer une plateforme d'hébergement et de diffusion de vidéos. Il se présente comme une alternative décentralisée à la plateforme YouTube. Une personne souhaitant avoir son instance de PeerTube peut l'installer sur un serveur. Une personne voulant simplement utiliser le service peut choisir parmis de nombreuses entrées [JoinPeerTube, 2018].

La décentralisation a été choisie pour de nombreuses raisons différentes. Tout d'abord, il y a une volonté à ne pas reproduire les mécanismes de centralisation qui, posent les nombreux problèmes présentés plus tôt. De plus, il est inimaginable de mettre en place une plateforme unique sans avoir les moyens financiers de Google. La décentralisation et la fédération permettent d'avoir des plateformes à taille humaine, un large catalogue de vidéos sans avoir à toutes les héberger. PeerTube implémente le protocole ActivityPub, il appartient donc à la fédivers : une personne utilisatrice de PeerTube peut communiquer des informations avec une personnes utilisatrice de Mastodon.

De plus, PeerTube met en place le pair-à-pair. Ce protocole est utilisé pour créer une forme de solidarité entre les personnes lors du visionnage de vidéos. En effet, si un serveur hébergeant les vidéos est trop sollicité, il peut tomber en panne et ne plus être accessible.

En règle générale, lorsqu'une personne visionne des vidéos sur Internet, celle-ci est partagée du serveur vers la personne. Sur PeerTube, en plus de recevoir les données du serveurs, la personne en partage une partie avec les personnes visionnant la même vidéos (figure C.2).

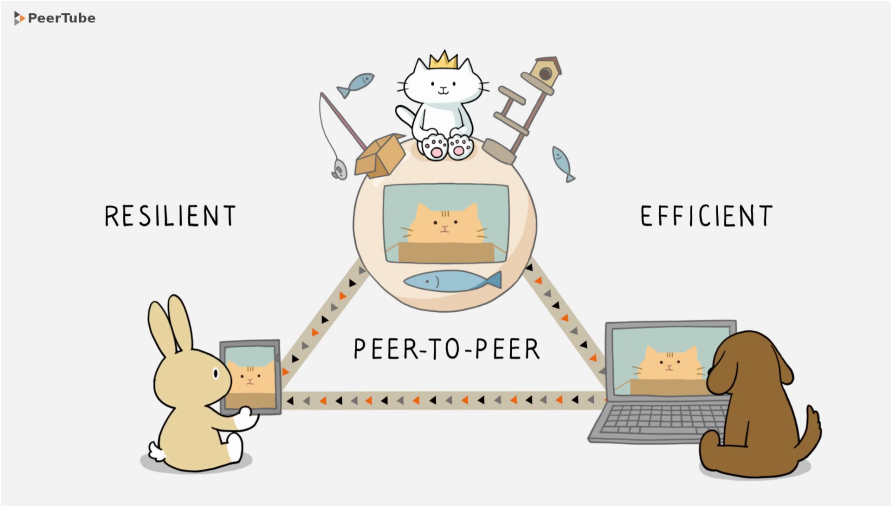


Figure C.2 – Illustration du pair-à-pair sur PeerTube, par l'association LILA

En juin-juillet 2018, Framasoft a lancé une campagne de financement afin de faire connaître le logiciel. Cette campagne est une réussite et permet d'officialiser la sortie de la première version de PeerTube à octobre 2018.

Une telle campagne a nécessité la vulgarisation du concept de fédération afin que les enjeux soient saisis par le plus grand nombre.

Annexe D

Cryptographie asymétrique

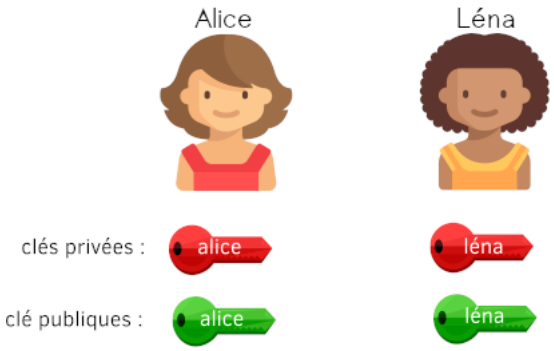


FIGURE D.1 – Le modèle de la cryptographie asymétrique

La cryptographie asymétrique est un domaine de la cryptographie [1] où il existe une distinction entre des données publiques et privées. Elle s'oppose à la cryptographie symétrique où les membres de confiance du réseau connaissent la clé secrète (unique) qui permet de décrypter les messages : cela peut s'assimiler à un langage codé entre plusieurs participants.

Dans la cryptographie asymétrique, chaque élément du réseau possède *une clé privée et une clé publique* (figure D.1). La clé publique d'un acteur est partagée avec toutes les personnes du réseau et la clé privée reste, comme son nom l'indique, privée. Un acteur ne doit jamais dévoiler sa clé privée.

De manière générale, une clé privée unique est générée aléatoirement pour chaque nouvel acteur du réseau. Un algorithme permet d'obtenir une clé publique unique à partir de cette clé privée. Cet algorithme ne doit pas pouvoir être inversé.

La cryptographie asymétrique est souvent utilisée pour répondre à deux problématiques différentes :

1. Discipline qui s'attache à protéger des messages.

— Je veux m'assurer que la personne qui m'envoie un message est bien celle qu'elle prétend être (D.1 Signature et vérification) ;
— Je veux m'assurer que personne ne peut lire les messages qui me sont destinés (D.2 Chiffrement et déchiffrement).

D.1 Signature et vérification

Dans cette utilisation de la cryptographie asymétrique, Léna utilise sa clé privée pour *signer* le message qu'elle veut envoyer à Alice. Tout le réseau connaît la clé publique de Léna donc tout le monde peut potentiellement lire son message : l'idée ici n'est donc pas de cacher le message.

L'intérêt est que si l'on utilise la clé publique de quelqu'un d'autre pour vérifier le message de Léna, cela ne donnera pas un résultat concluant. Si après vérification avec la clé publique de Léna le message est cohérent, cela nous assure que ce message vient d'elle.

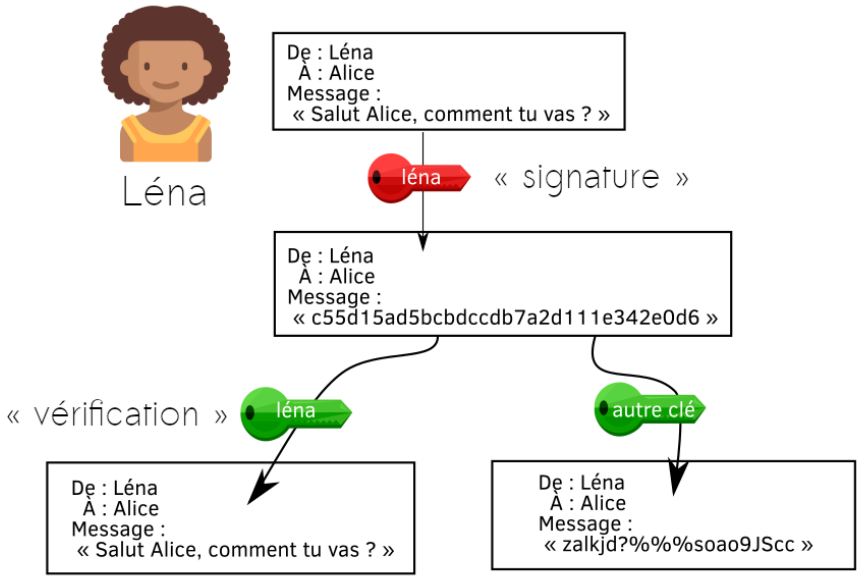


Figure D.2 – Schéma du système de signature - vérification

Cette utilisation est intéressante dans les médias sociaux où une grande partie des messages sont publics : cela empêche qu'une personne tierce se fasse passer pour quelqu'un d'autre.

C'est cet aspect de la cryptographie asymétrique qui est utilisé dans l'implémentation

présentée du protocole ActivityPub.

D.2 Chiffrement et déchiffrement

Dans cette utilisation de la cryptographie asymétrique, Alice utilise la clé publique de Léna pour chiffrer un message qu'elle souhaite lui envoyer. Une fois le message chiffré, seule la clé privée de Léna peut le déchiffrer donc seule Léna peut lire les messages qui lui sont destinés. Même Alice ne peut pas déchiffrer son message une fois qu'il a été chiffré.

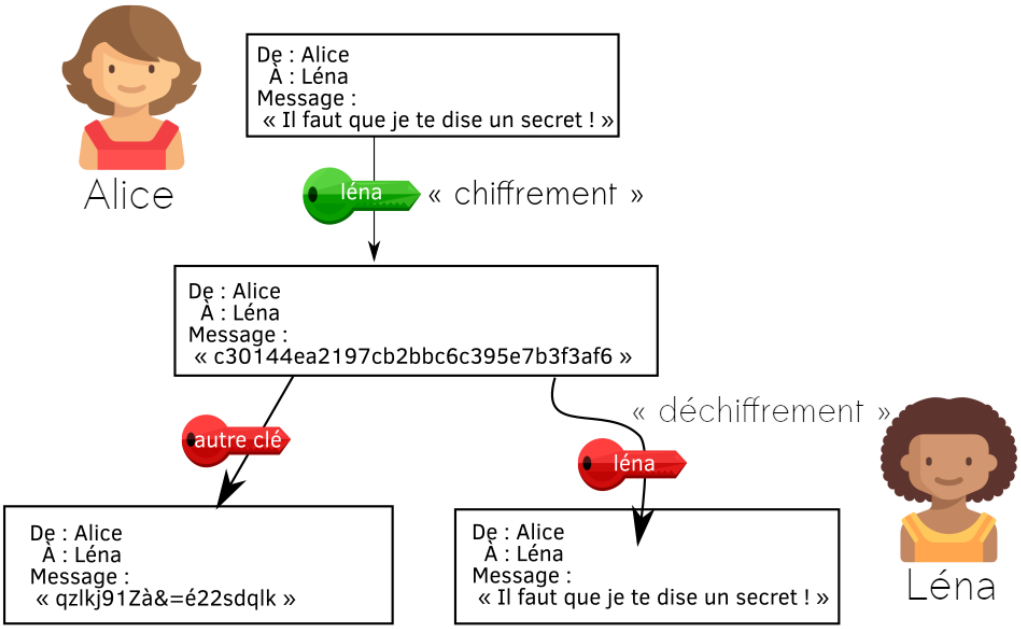


Figure D.3 – Schéma du système de chiffrement-déchiffrement

D.3 Signature des requêtes HTTP

La signature des requêtes HTTP – en anglais HTTP signature [IETF,], est un protocole standardisé par l'IETF. Dans les communications sur Internet, il permet de répondre à la problématique suivante :

— Je veux être sûr que la personne qui m'envoie un message est bien celle qu'elle prétend être. L'usurpation d'identité ne doit pas être possible ;

Pour ce faire, le protocole utilise la méthode de cryptographie asymétrique pour signer la requête.

```
1    var httpSignatureOptions = {
2     algorithm: 'rsa-sha256',
3     authorizationHeaderName: 'Signature',
4     keyId,
5     key: **PRIVATE KEY OF SENDER**
6   };
7
8
9   var objectOptions = {
10    url: ** RECIPIENT INBOX URL **,
11    json: true,
12    method: 'POST',
13    body: ** JSON OBJECT TO SEND **,
14    headers: {
15      'Accept': 'application/activity+json'
16    },
17    httpSignature: httpSignatureOptions
18  };
19
20
21  request(objectOptions);
```

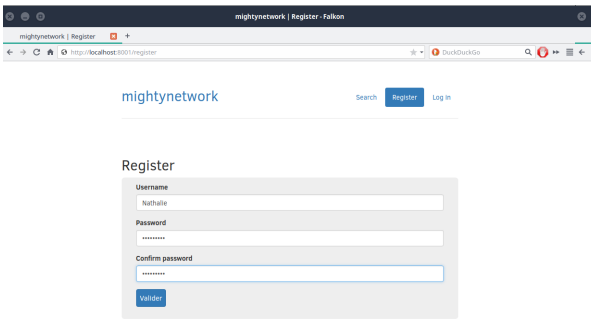
Listing D.1 – Signature d'une requête HTTP

À ligne `21` du listing D.1 se trouve l'expression de la requête HTTP qui prend en entée l'objet `objectOptions`, défini à la ligne 9. Cet objet précise les nombreuses options de la requête à effectuer. La signature de la requête s'effectue à la ligne `17`. Le options de cette signature sont précisés dans la variable `httpSignatureOptions`, lignes `1-6`.

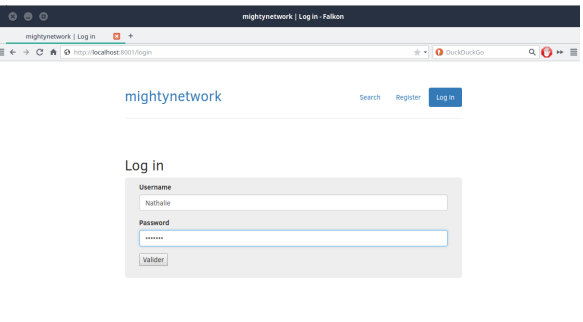
Pour respecter le protocole HTTP Signature, il faut que l'émetteur et le récepteur de la requête s'accordent sur ces options : l'algorithm de signature, le nom de l'en-tête à accepter, la clé `keyId`, qui est spécifique à chaque acteur envoyant un message. Cette clé est signée par la clé privée de l'émetteur et vérifiée par sa clé publique par le récepteur. Elle est décidée par les deux parties et ne doit pas changer car elle permet la vérification expliquée en annexe D.1.

Ainsi, contrairement à l'exemple de la figure D.2, ce n'est pas le message qui est signé mais un autre objet présent dans l'en-tête qui est spécifique à chaque membre du réseau et connu de tous. Dans le protocole ActivityPub, la clé utilisée pour cette vérification est l'adresse Webfinger (section 3.2.4) dans le réseau : `acct:lena@uneinstance.org`.

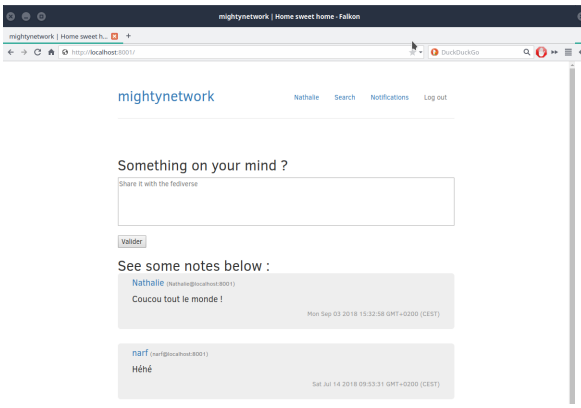
Annexe E



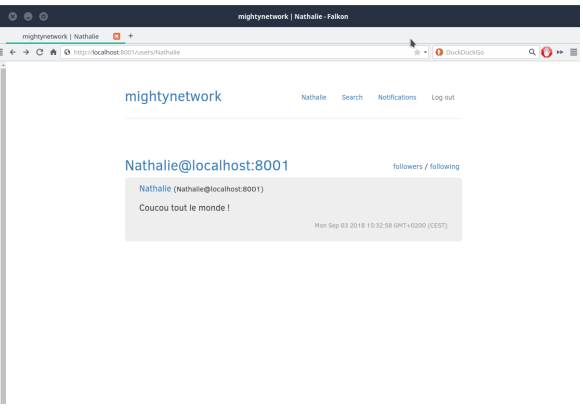
(a) Page d'inscription



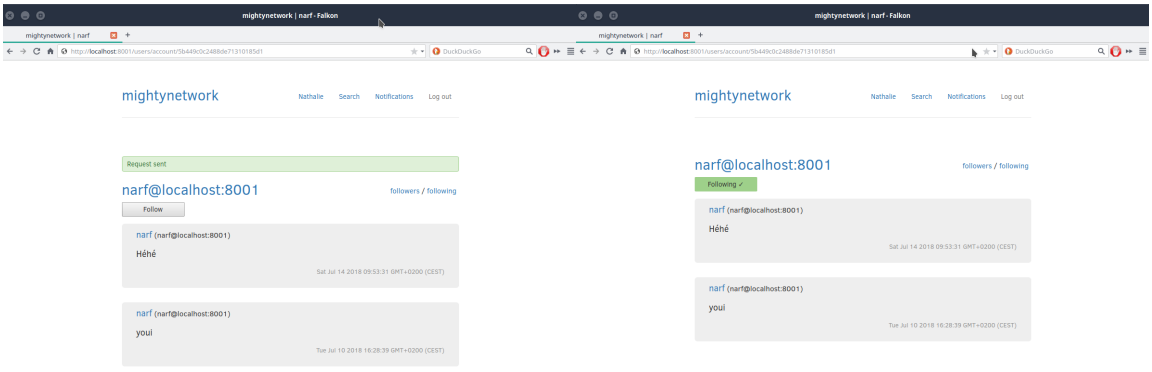
(b) Page de connexion



(c) Page d'accueil

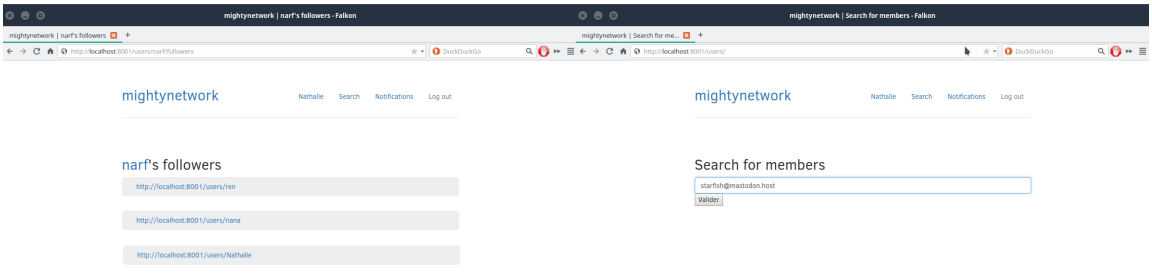


(d) Page de l'utilisatrice



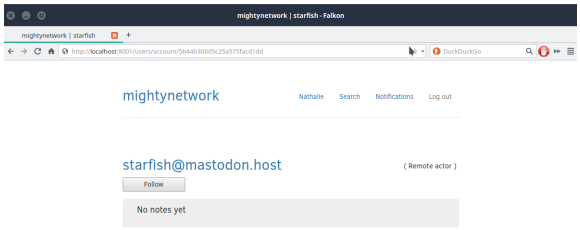
(a) Page d'une utilisatrice local après l'envoi d'une demande de *Follow*

(b) Page d'une personne suivie par l'utilisatrice en cours



(c) Page des personnes suivant *narf*

(d) Page de recherche d'acteurs de la fediverse



(e) Page d'un acteur distant

Figure E.1 – Quelques captures d'écran de la preuve de concept réalisée