AmarOk

Geste de savoir

p2p internals #4 - Disséquons git

27 mars 2021

Table des matières

Continuons cette série d'articles sur les systèmes distribués en parlant d'un outil que beaucoup utilisent chaque jour : *git*.

Git est un outil de versionnage. Contrairement à certains systèmes tels que *subversion*, *git* ne dépend pas d'un serveur. Chaque personne possède une copie plus ou moins partielle du projet qui peut être transmise d'une personne à une autre directement. Alors, même si aujourd'hui une grande partie des gens utilisent des systèmes comme Gitea , GitLab , GitHub , etc. il est aussi tout à fait possible de faire sans (en cas de panne de l'hébergeur par exemple).

Dans cet article, je ne veux **pas** expliquer **comment** utiliser *git*. Des milliers d'articles à ce propos existent déjà (exemple: https://try.github.io/), et si tu cherches à savoir comment utiliser *git*, cet article n'est pas pour toi. Mais je vais essayer d'expliquer ce qu'il se passe lorsque quelqu'un tape `git clone https://github.com/zestedesavoir/zmarkdown/` dans son terminal d'un point de vue réseau.

Pour se faire, nous allons réaliser un petit programme (en *Rust*, mais ce n'est pas vraiment important) qui va cloner un dépot sur un protocole inventé pour les besoins de cet article (qu'on appellera WOLF[1]).

1. Un transport git c'est quoi ?

1.1. Les différentes couches du transport

Pour échanger des données entre un serveur et un client, *git* utilise des *packfiles* transférées via ce qu'on appelle le `pack-protocol` ou le `http-protocol`. Ces deux protocoles ont des logiques relativement similaires, mais dans la suite de ce billet, on regardera seulement le `pack-protocol`. La définition du `http-protocol` étant laissé à la curiosité du lecteur (mais sa documentation est encore incomplète).

1. *git* transmet des pack-files. On peut donc parler de wolf-pack

Ces deux protocoles sont composés de deux services:

1. `upload-pack` (côté serveur, `fetch-pack` côté client) qui permet de transférer des données du serveur vers le client. On se concentrera sur ce service dans la suite du billet.
2. `received-pack` (côté serveur, `send-pack` côté client) pour envoyer des données d'un client à un serveur.

Finalement, ces protocoles sont utilisés par dessus différents transports pour transmettre les paquets. De base, il est possible d'utiliser 4 types de transports : `ssh://` (SSH), `file://` (un pipe), `git://` (un simple serveur TCP , généralement combiné avec SSH pour l'authentification) pour le `pack-protocol` et `http://` pour le `http-protocol`.

Les protocoles HTTP sont de 2 types. Les `Dumb` qui sont de simples serveurs web proposant des fichiers, et les `smart` répondant à des services *git*.

Pour résumer, voici ce que j'appelle un transport *git*:

`upload-pack` OU `receive-pack`	
`pack-protocol` + `pack-format`	`http-protocol` + `pack-format`
Protocole de communication tel que `ssh://`, `file://`, `git://`.	`http://` ou `https://`.

Et voici ce que nous allons réaliser :

	`upload-pack`
`pack-protocol` + `pack-format`	
`wolf://`	

Finalement les protocoles *git* définissent deux grandes notions, le format `pkt-line` et le format `pack`.

1.1.1. Le format pkt-line

Une `pkt-line` est une chaîne **binaire**. Les 4 premiers octets de cette chaîne sont utilisés pour stocker la taille totale de la ligne au format hexadécimal. Ainsi, une `pkt-line` débutant par 0023 fera donc 35 octets de long.

Le format a quelques autres spécificités comme:

— Si la chaîne est non binaire (ne contient pas de `\0`) elle devrait finir par un `LF` (`\n`) qui doit être inclu dans la taille.
— Il est interdit d'envoyer un paquet vide (0004). Ce paquet vide est appellé *flush-pkt* est s'écrit 0000.

— La taille maximale d'un `pkt-line` est de 65520 au total (65516 de données).

Exemple, pour envoyer "foobar\n" on envoie "000bfoobar\n".

1.1.2. Le format pack

Une fois la partie négociation réalisée. Il est alors nécessaire de transmettre les objets. Ces objets sont transférés via le format *pack* qui peut faire l'objet d'un article complet. Mais comme de nombreuses bibliothèques (comme *libgit2* que nous allons utiliser) fournissent déjà toute l'abstraction nécessaire pour générer des données dans ce format, je ne vais pas m'attarder dessus.

Cependant, si vous souhaitez comprendre exactement ce que contient un *packfile*, je vous recommande la lecture de cette page: https://git-scm.com/docs/pack-format

Pour résumer, voici généralement à quoi ressemble le contenu de ce format:

	En-tête			**données**		**checks**
exemple	signature (PACK)	version (2)	nombre d'objets	type	contenu	trailer

Les données qui peuvent être transférées sont généralement de 4 types possibles (*commit, tree, tag, blob*) et peuvent être compressées dans un format appellé *Deltified representation*.

1.2. La commande inutile donc indispensable : upload-pack

git est une commande qui possède une très longue liste de sous-programmes. Certains ne vous serviront (heureusement) jamais mais elles possèdent tout de même un petit intérêt; comme par exemple `git http-backend` permettant de rouler rapidement un serveur *git* accessible en HTTP (et donc de jouer avec le `http-protocol` que je mentionnais). Cependant, certaines de ces commandes n'existent que pour être invoquées par d'autres (mais **jamais** par l'utilisateur). C'est par exemple le cas de la commande que je vous propose de découvrir, `git upload-pack` qui va nous permettre de jouer avec le `pack-protocol`.

> *i* `git upload-pack` est une commande invoquée par `git fetch-pack` utilisée par `git fetch`. En d'autres termes, `git upload-pack` permet de faire un `git fetch` ou `git clone` de la manière la plus manuelle possible.

Ainsi, voici à quoi ressemble l'utilisation de `git upload-pack` sur le repo de ZMarkdown :

```
1 # https://github.com/zestedesavoir/zmarkdown a été cloné pour
      simplifier la trace.
2 # via `git clone git@github.com:zestedesavoir/zmarkdown.git`
3 # 7e31d5dcc19fa412df5674676f5fb7092d035275 est le commit pointé par
      master au moment où j'écris.
```

```
4
5 $ echo
      "0032want 7e31d5dcc19fa412df5674676f5fb7092d035275\n00000009done\n"
      | git upload-pack zmarkdown > dump_git_upload_pack
```

Pour résumer ce que fait cette commande, elle demande de récupérer tout l'arbre de commits situé en dessous du commit $7e31d5dcc19fa412df5674676f5fb7092d035275$ et d'écrire tout le contenu entrant dans le fichier *dump_git_upload_pack*.

Ce fichier servira de base pour la seconde partie mais n'est pas important à regarder. Par contre si un jour vous avez besoin de debugguer ce qu'un serveur `git` vous envoie, je peux conseiller de le lire avec `xxd dump_git_upload_pack | less` ou `less dump_git_upload_pack` pour les premières lignes.

2. Réalisons notre propre serveur git

Maintenant que nous possèdons les bases de comment fonctionne *git*, il est temps de mettre ces connaissances en pratique et de creuser un peu en réalisant un petit serveur *git* qui permettra de répondre à un `git clone` ou `git fetch` sur un protocole custom.

Le code complet sera disponible dans la partie suivante.

2.1. Découverte des références

Ainsi, comme nous l'avons vu, la première chose que réalisera notre client est d'envoyer la commande `git-upload-pack`. Le format tel que précisé par la documentation est le suivant:

pkt_len (4 octets)	git-upload-pack	space	path-name	\0	host-parameter	\0	\0	extra-paramete

Par exemple, notre client va envoyer

```
1 003dgit-upload-pack zmarkdown.git\0host=localhost\0\0version=1\0
```

À noter que la partie *extra-parameters* est optionelle et ne supporte que `version=1` pour le moment. Donc ceci est valide:

```
0032git-upload-pack zmarkdown.git\0host=localhost\0
```

Puis le travail de notre serveur va commencer en envoyant ses références. Ainsi, voici à quoi va commencer à ressembler notre code:

```rust
impl Server {
    // La boucle principale de notre server, il lit juste ce que le client envoie
    // (qui sera un vecteur d'octets (Vec<u8>)
    pub fn read(&mut self) {
        loop {
            let buf = self.channel.lock().unwrap().recv().unwrap();
            self.recv(buf);
        }
    }

    fn recv(&mut self, buf: Vec<u8>) {
        let mut buf = Some(buf);
        let mut need_more_parsing = true;
        while need_more_parsing {
            need_more_parsing = self.parse(buf.take());
        }
    }

    // Fonction qui servira à traiter les données entrantes
    fn parse(&mut self, mut buf: Option<Vec<u8>>) -> bool {
        // Le client pouvant envoyer plusieurs pkt-line, self.buf sert de cache.
        if buf.is_some() {
            self.buf.append(&mut buf.unwrap());
        }
        // Les 4 premiers octets définissent une taille (>4, 0000 étant un FLUSH)
        let pkt_len = str::from_utf8(&self.buf[0..4]).unwrap();
        let pkt_len = max(4 as usize, i64::from_str_radix(pkt_len, 16).unwrap() as usize);
        let pkt : Vec<u8> = self.buf.drain(0..pkt_len).collect();
        let pkt = str::from_utf8(&pkt[0..pkt_len]).unwrap();
        println!("Received pkt: {}", pkt);

        if pkt.find(UPLOAD_PACK_CMD) == Some(4) {
            // Cf: https://github.com/git/git/blob/master/Documentation/technical/pack-protocol.txt#L166
            // Notre git-upload-pack est détecté. Pour simplifier on ignore les *extra-parameters*
            println!("Upload pack command detected");
```

```
36              // Envoyons nos références
37              self.send_references_capabilities();
38          }
39          // Si d'autres pkt-line doivent être traitées, le cache ne
        sera pas vide
40          self.buf.len() != 0
41      }
42  }
```

Maintenant, concentrons nous sur `self.send_references_capabilities();`. Voici ce que nous dit la référence:

- Tout d'abord le serveur doit répondre par sa version si `version` est passée dans les *extra-parameters* (pas le cas ici, mais le paquet attendu est $000eversion1$),
- Puis la liste de toutes les références connues et des hashs pointés,
- La liste des références est ordonnée,
- Si HEAD est présent, alors HEAD doit être la première référence affichée,
- La première référence est suivie par `\0` puis les capacités supportées par le serveur.
- La liste est terminée par un paquet `FLUSH`.

Ce qu'on peut donc traduire par:

```
1  fn send_references_capabilities(&self) {
2      let current_head =
       self.repository.refname_to_id("HEAD").unwrap();
3      let mut capabilities =
       format!("{} HEAD\0side-band side-band-64k shallow no-progress include-tag"
       current_head);
4      capabilities = format!("{:04x}{}\n", capabilities.len() + 5 /*
       taille + \n */, capabilities);
5
6      for name in self.repository.referencess().unwrap().names() {
7          let reference : &str = name.unwrap();
8          let oid =
       self.repository.refname_to_id(reference).unwrap();
9          capabilities += &*format!("{:04x}{} {}\n", 6 /* taille +
       espace + \n */ + 40 /* oid */ + reference.len(), oid,
       reference);
10     }
11
12     print!("Send: {}", capabilities);
13
       self.channel.lock().unwrap().send(capabilities.as_bytes().to_vec()).unwrap
14     println!("Send: {}", FLUSH_PKT);
15
       self.channel.lock().unwrap().send(FLUSH_PKT.as_bytes().to_vec()).unwrap();
16 }
```

Le serveur peut proposer énormément d'options que je ne vais pas détailler ici. Mais par exemple, si le client demande à l'étape suivante `side-band` et `side-band-64k`, le serveur va devoir envoyer les données sous format multiplexé, de maximum 999 octets pour `side-band`, 65519 pour `side-band-64k` + 1 octet de contrôle (je reviendrais dessus). `no-progress` fait en sorte que le serveur n'envoie pas la progression. Le reste des propriétés sont décrites ici .

Finalement, voici à quoi ressemble notre début de communication:

```
C: 0028git-upload-pack /zdshost=localhost\0host=localhost\0
S: 006ac29eb686c3638633bff5b852ee1ed76211882ba1 HEAD\0side-band
    side-band-64k shallow no-progress include-tag
S: 003f7300c6f39366e2c0bcaf99efc05cc45e8511a384 refs/heads/master
S: 0046c29eb686c3638633bff5b852ee1ed76211882ba1
    refs/remotes/origin/HEAD
S: 0048c29eb686c3638633bff5b852ee1ed76211882ba1
    refs/remotes/origin/master
S: 0000
```

Maintenant que le client sait ce que le serveur peut proposer, il lui est possible de demander ce qu'il souhaite. C'est parti pour la partie négociation!

2.2. Négociation du Packfile

Si le client ne souhaite rien (par exemple si il ne cherche qu'à réaliser un `ls-remote`) il peut terminer directement la connexion via un `FLUSH`, mais généralement le client va rentrer dans une phase de négociation. Le client va alors annoncer les commits qu'il souhaite, possède, et les options supportées.

Le client doit au moins envoyer une ligne `want`, et les commits référencés doivent être ceux proposés par le serveur à la phase précédente. Si le client possède des objets dont il possède une *shallow copy* il doit le dire. Il annonce aussi la profondeur maximale de l'historique souhaitée. Ensuite, et dans le but d'obtenir le *packfile* minimum le client envoie la liste des (au maximum 32) commits qu'il possède.

Puis le serveur termine en envoyant le `ACK` pour les objets qu'il possède. Pour se faire, trois modes existent.

- Le `multi-ack` où:
  - Il répond 'ACK obj-id continue' pour les commits communs.
  - `ACK` pour tous les commits une fois qu'un commit commun est trouvé
  - `NAK` pour attendre une nouvelle réponse du client (`done` ou de nouveaux `have`).
- Le `multi-ack-detailed`:
  - `ACK obj-id ready` et `ACK obj-id common` peuvent être envoyés
- sinon:
  - `ACK obj-id` sur le premier objet commun puis plus rien tant que le client n'envoie pas `done`.
  - `NAK` si il reçoit un paquet `FLUSH` est qu'aucun paquet commun n'a été trouvé.

Le client décide alors si les informations données par le serveur sont suffisantes et peut alors soit redemander des informations en complétant sa liste de `have` (jusqu'à 256 `have` (sinon le serveur enverra juste tous ces objets)) et envoie un `done` lorsqu'il a terminé.

Le serveur termine cette phase en envoyant soit un dernier `NAK` (si pas de commit commun) ou un `ACK obj-id` final.

Ce qui nous donne par exemple:

Pour un clone:

```
C: 004dwant c42412dfee248e8acbd21df6038991dff6fd1b2a side-band-64k
    include-tag\n
C: 0000
C: 0009done\n
S: 0008NAK\n
```

Ou pour un fetch:

```
C: 004dwant c42412dfee248e8acbd21df6038991dff6fd1b2a side-band-64k
    include-tag\n
C: 0000
C: 0032have eafc4105f398a014c97e0db4b171d787aeeef0d7\n
C: 0000
S: 003aACK c42412dfee248e8acbd21df6038991dff6fd1b2a continue\n
S: 0008NAK\n
C: 0009done\n
S: 0031ACK c42412dfee248e8acbd21df6038991dff6fd1b2a\n
```

2.3. Envoi des données

Finalement, la dernière étape est d'envoyer les objets dans un *packfile*. `libgit2` possède l'objet `PackBuilder` qui va être utile pour préparer les données. De notre côté, il suffit juste d'ajouter les commits attendus. On va donc ajouter tous les commits se situant entre le commit souhaité (via le `want`) jusqu'au commit que le client possède forcément (si des *merge commits* sont trouvés, il faut parcourir les différentes branches mergées jusqu'à la racine commune).

Une subtilité existe tout de même pour l'envoi des données. Le serveur que nous réalisons offre la capacité `side-band-64k`. Ce qui signifie que notre *pack-file* sera découpé en morceaux de 64k. Le paquet sera donc du format décrit un peu plus haut: 4 octets pour la taille, 1 pour l'octet de contrôle ($0x1$ ici car pour les données. $0x2$ est utilisé pour envoyer la progression $0x3$ pour les erreurs), 65515 octets de données.

Et le tout est terminé par un paquet `FLUSH`.

Ce qui nous donne:

```rust
fn send_pack_data(&self) {
        println!("Send: [PACKFILE]");
        // Note: Ici, on ajoute tous les commits tant qu'on à pas
        // trouvé un commit annoncé par le client et qu'on ne sois
        // pas sûr d'avoir parcouru toutes les branches (ou jusqu'au
        // commit initial).
        let mut pb = self.repository.packbuilder().unwrap();
        let fetched = Oid::from_str(&*self.wanted).unwrap();
        let mut revwalk = self.repository.revwalk().unwrap();
        let _ = revwalk.push(fetched);
        let _ = revwalk.set_sorting(Sort::TOPOLOGICAL);

        let mut parents : Vec<String> = Vec::new();
        let mut have = false;

        while let Some(oid) = revwalk.next() {
            let oid = oid.unwrap();
            let oid_str = oid.to_string();
            have |= self.have.iter().find(|&o| *o == oid_str).is_some();
            if let Some(pos) = parents.iter().position(|p| *p == oid_str) {
                parents.remove(pos);
            }
            if have && parents.is_empty() {
                // Commit initial
                break;
            }
            let _ = pb.insert_commit(oid);
            let commit = self.repository.find_commit(oid).unwrap();
            let mut commit_parents = commit.parents();
            // On doit être sûr de parcourir tous les chemins
            while let Some(p) = commit_parents.next() {
                parents.push(p.id().to_string());
            }
        }

        // Note: Packbuilder possède des fonctions pour écrire des chunks et pas tout le packfile d'un coup.
        let mut data = Buf::new();
        let _ = pb.write_buf(&mut data);

        let len = data.len();
        let data : Vec<u8> = data.to_vec();
        let mut sent = 0;
        while sent < len {
```

```
44              // cf
    https://github.com/git/git/blob/master/Documentation/technical/pack-
    protocol.txt#L166
45              // Si side-band-64k est spécifié, on doit envoyer
    jusqu'à 65519 octets + 1 de contrôle par pkt-line.
46              let pkt_size = min(65515, len - sent);
47              // Le paquet: Size (4 octets), Control byte (0x01 pour
    les données), pack data.
48              let pkt = format!("{:04x}", pkt_size + 5 /* taille +
    control */);
49
    self.channel.lock().unwrap().send(pkt.as_bytes().to_vec()).unwrap();
50
    self.channel.lock().unwrap().send(b"\x01".to_vec()).unwrap();
51
    self.channel.lock().unwrap().send(data[sent..(sent+pkt_size)].to_vec()).unw
52              sent += pkt_size;
53          }
54
55          println!("Send: {}", FLUSH_PKT);
56          // Et on fini par un FLUSH
57
    self.channel.lock().unwrap().send(FLUSH_PKT.as_bytes().to_vec()).unwrap();
58      }
```

Et voilà, le client a récupéré les données souhaitées, il peut maintenant fermer la connexion!

3. Testons notre serveur

Notre serveur minimal est dorénavant opérationel. On peut le tester avec un vrai répertoire à cloner.

Le petit programme de test que je propose se décompose en 3 parties. La première partie décrit le transport pour libgit2 . Cette bibliothèque est très pratique et pas mal utilisée pour réaliser différents programmes utilisant *git*. Elle possède un wrapper *Rust* que je vais utiliser ici: git2-rs . Le problème est que la documentation est incomplète, surtout pour la partie parlant des *smart-transports*, donc je vous invite à trouver des exemples implémentant ce que vous souhaitez si un jour vous en avez besoin. En tout cas, voici mon fichier `src/wolftransport.rs` qui définit un *smart-transport* utilisant le protocole WOLF. Ce protocole ajoute devant tous les paquets envoyés un *header* de 4 bytes : $WOLF$.

`src/wolftransport.rs`:

Contenu masqué n°1

Puis vient notre server `src/server.rs`:

Contenu masqué n°2

Puis notre fichier `src/main.rs` pour lancer le tout:

Contenu masqué n°3

Et maintenant lançons le programme qui copie un petit répertoire *git* d'un dossier à un autre:

```
#Note, comme le serveur ne supporte pas le multi-ack, on ne copiera que la branche master (sans les tags)
 amarok@tars3 ▶ ~/Projects/p2p-internals-git ▶  main ▶ git clone -b master --single-branch --no-tags https://github.com/zestedesavoir/zmarkdown.git
Cloning into 'zmarkdown'...
remote: Enumerating objects: 1207, done.
remote: Counting objects: 100% (1207/1207), done.
remote: Compressing objects: 100% (206/206), done.
remote: Total 16609 (delta 1006), reused 1192 (delta 1001), pack-reused 15402
Receiving objects: 100% (16609/16609), 10.85 MiB | 2.25 MiB/s, done.
Resolving deltas: 100% (11966/11966), done.
# On copie via notre petit outil
 amarok@tars3 ▶ ~/Projects/p2p-internals-git ▶  main ▶ ./target/debug/p2p-internals-git zmarkdown zmarkdown-copy
Starting server for zmarkdown
RECV: 0028git-upload-pack /zdshost=localhost
Upload pack command detected
Send: 006ac42412dfee248e8acbd21df6038991dff6fd1b2a HEADside-band side-band-64k shallow no-progress include-tag
Send: 003fc42412dfee248e8acbd21df6038991dff6fd1b2a refs/heads/master
Send: 0046c42412dfee248e8acbd21df6038991dff6fd1b2a refs/remotes/origin/HEAD
Send: 0048c42412dfee248e8acbd21df6038991dff6fd1b2a refs/remotes/origin/master
Send: 0000
RECV: 004dwant c42412dfee248e8acbd21df6038991dff6fd1b2a side-band-64k include-tag

Detected wanted commit: c42412dfee248e8acbd21df6038991dff6fd1b2a
RECV: 0000
RECV: 0009done

Peer negotiation is done. Answering to want order
```

```
Send: 0008NAK
Send: [PACKFILE]
Send: 0000
Cloned into "zmarkdown-copy"!
# Et plus qu'à valider!
 amarok@tars3 ▶ ~/Projects/p2p-internals-git ▶ ᚠ main ▶ git -C
    zmarkdown rev-parse HEAD
c42412dfee248e8acbd21df6038991dff6fd1b2a
 amarok@tars3 ▶ ~/Projects/p2p-internals-git ▶ ᚠ main ▶ git -C
    zmarkdown-copy rev-parse HEAD
c42412dfee248e8acbd21df6038991dff6fd1b2a
```

Et voilà, on a réalisé un petit serveur *git* qui clone un répertoire. Bien entendu ce serveur est loin d'être complet et beaucoup de choses manquent, mais j'espère vous avoir éclairé sur comment *git* communique entre son client et son serveur.

3.1. Références

Merci de m'avoir lu jusqu'ici. Voici une liste de références qui m'ont servi durant l'écriture de cet article ou pour des projets personnels:

— https://libgit2.org/libgit2/#HEAD (**!! Attention, des pans de la documentation manquent. Comme celle sur les *smart-transports* !!**)
— https://git-scm.com/docs/ & https://github.com/git/git/blob/master/Documentation/technical/ qui décrivent comment les protocoles utilisés par *git* fonctionnement.
— https://docs.rs/git2/0.13.17/git2/ pour la documentation de la bibliothèque que j'utilise.
— https://github.com/rust-lang/git2-rs/tree/master/git2-curl et https://github.com/libgit2/libgit2/blob/main/src/transports/git.c qui implémentent des transports *git*
— Le code de ce billet: https://github.com/AmarOk1412/p2p-internals-git/

Contenu masqué

Contenu masqué n°1

```rust
// https://docs.rs/git2/0.13.17/git2/transport/fn.register.html

use bichannel::{ SendError, RecvError };
use git2::Error;
use git2::transport::SmartSubtransportStream;
use git2::transport::{Service, SmartSubtransport, Transport};
use std::io;
```

Contenu masqué

```rust
use std::io::prelude::*;
use std::sync::{Arc, Mutex};

static HOST_TAG: &str = "host=";

// Note : Ce transport est purement fictif et ne fait que ajouter un header
// de 4 octets: "WOLF"
pub struct WolfChannel
{
    pub channel: bichannel::Channel<Vec<u8>, Vec<u8>>,
}

impl WolfChannel
{
    pub fn recv(&self) -> Result<Vec<u8>, RecvError> {
        let res = self.channel.recv();
        if !res.is_ok() {
            return res;
        }
        let mut res = res.unwrap();
        res.drain(0..4);
        Ok(res)
    }

    pub fn send(&self, data: Vec<u8>) -> Result<(), SendError<Vec<u8>>> {
        let mut to_send = "WOLF".as_bytes().to_vec();
        to_send.extend(data);
        self.channel.send(to_send)
    }
}

pub type Channel = Arc<Mutex<WolfChannel>>;

// Maintenant, écrivons notre smart transport pour git2-rs répondant au scheme wolf://
struct WolfTransport {
    channel: Channel,
}

struct WolfSubTransport {
    action: Service,
    channel: Channel,
    url: String,
    sent_request: bool
}

pub unsafe fn register(channel: Channel) {
```

Contenu masqué

```rust
    git2::transport::register("wolf", move |remote| factory(remote, channel.clone())).unwrap();
}

fn factory(remote: &git2::Remote<'_>, channel: Channel) -> Result<Transport, Error> {
    Transport::smart(
        remote,
        false, // rpc = false, signifie que notre channel est connecté durant tout le long de la transaction
        WolfTransport {
            channel
        },
    )
}

impl SmartSubtransport for WolfTransport {
    /**
     * Génère un nouveau transport à utiliser. Comme rpc = false, on ne répond
     */
    fn action(
        &self,
        url: &str,
        action: Service,
    ) -> Result<Box<dyn SmartSubtransportStream>, Error> {
        Ok(Box::new(WolfSubTransport {
            action,
            channel: self.channel.clone(),
            url: String::from(url),
            sent_request: false,
        }))
    }

    fn close(&self) -> Result<(), Error> {
        Ok(())
    }
}

impl WolfTransport {
    fn generate_request(cmd: &str, url: &str) -> Vec<u8> {
        // url format = wolf://host/repo
        // Note: Cette requête n'est envoyé que quand le client démarre sa partie afin de notifier le serveur
        let sep = url.rfind('/').unwrap();
        let host = url.get(7..sep).unwrap();
        let repo = url.get(sep..).unwrap();

        let null_char = '\0';
        let total = 4                                   /* 4 bytes for the len len */
```

Contenu masqué

```
                    + cmd.len()                             /* followed by the command */
                    + 1                                     /* space */
                    + repo.len()                            /* repo to clone */
                    + 1                                     /* \0 */
                    + HOST_TAG.len() + host.len()           /* host=server */
                    + 1                                     /* \0 */;
        let request = format!("{:04x}{} {}{}{}{}{}", total, cmd, repo, null_char, HOST_TAG, host, null_char);
        request.as_bytes().to_vec()
    }
}

impl Read for WolfSubTransport {
    fn read(&mut self, buf: &mut [u8]) -> io::Result<usize> {
        // Envoie de la requête pour le serveur
        if !self.sent_request {
            let cmd = match self.action {
                Service::UploadPackLs => "git-upload-pack",
                Service::UploadPack => "git-upload-pack",
                Service::ReceivePackLs => "git-receive-pack",
                Service::ReceivePack => "git-receive-pack",
            };
            let cmd = WolfTransport::generate_request(cmd, &*self.url);
            let _ = self.channel.lock().unwrap().send(cmd);
            self.sent_request = true;
        }
        // Lis la réponse du serveur
        let mut recv = self.channel.lock().unwrap().recv().unwrap_or(Vec::new());
        let mut iter = recv.drain(..);
        let mut idx = 0;
        while let Some(v) = iter.next() {
            buf[idx] = v;
            idx += 1;
        }
        Ok(idx)
    }
}

impl Write for WolfSubTransport {
    fn write(&mut self, data: &[u8]) -> io::Result<usize> {
        let _ = self.channel.lock().unwrap().send(data.to_vec());
        Ok(data.len())
    }
    fn flush(&mut self) -> io::Result<()> {
        // Non utilisé dans notre cas
```

```
143         Ok(())
144     }
145 }
```

Retourner au texte.

Contenu masqué n°2

```rust
use crate::wolftransport::Channel;
use git2::{ Buf, Oid, Repository, Sort };
use std::cmp::{ max, min };
use std::i64;
use std::str;

static FLUSH_PKT: &str = "0000";
static NAK_PKT: &str = "0008NAK\n";
static DONE_PKT: &str = "0009done\n";
static WANT_CMD: &str = "want";
static HAVE_CMD: &str = "have";
static UPLOAD_PACK_CMD: &str = "git-upload-pack";

/**
 * Représente un serveur git fonctionnant sur notre serveur personnalisé servan
 */
pub struct Server {
    repository: Repository,
    channel: Channel,
    wanted: String,
    common: String,
    have: Vec<String>,
    buf: Vec<u8>,
    stop: bool,
}

impl Server {
    pub fn new(channel: Channel, path: &str) -> Self {
        let repository = Repository::open(path).unwrap();
        Self {
            repository,
            channel,
            wanted: String::new(),
            common: String::new(),
            have: Vec::new(),
            buf: Vec::new(),
            stop: false,
        }
```

Contenu masqué

```rust
    }

    pub fn run(&mut self) {
        // Stop est mis à true quand le clone est terminé.
        while !self.stop {
            let buf = self.channel.lock().unwrap().recv().unwrap();
            self.recv(buf);
        }
    }

    fn recv(&mut self, buf: Vec<u8>) {
        let mut buf = Some(buf);
        let mut need_more_parsing = true;
        // Comme le client peut envoyer plusieurs pkt-line, on traite
        // jusqu'à ce qu'on soit prêt à recevoir de nouveaux ordres.
        while need_more_parsing {
            need_more_parsing = self.parse(buf.take());
        }
    }

    fn parse(&mut self, buf: Option<Vec<u8>>) -> bool {
        // Parse la taille du pjt-line
        // Référence : https://github.com/git/git/blob/master/Documentation/technical/protocol-common.txt#L51
        // Les 4 premiers octets stockent la taille, sauf pour 0000 qui est le paquet FLUSH
        if buf.is_some() {
            self.buf.append(&mut buf.unwrap());
        }
        let pkt_len = str::from_utf8(&self.buf[0..4]).unwrap();
        let pkt_len = max(4 as usize, i64::from_str_radix(pkt_len, 16).unwrap() as usize);
        let pkt : Vec<u8> = self.buf.drain(0..pkt_len).collect();
        let pkt = str::from_utf8(&pkt[0..pkt_len]).unwrap();
        println!("RECV: {}", pkt);

        if pkt.find(UPLOAD_PACK_CMD) == Some(4) {
            // Cf: https://github.com/git/git/blob/master/Documentation/technical/pack-protocol.txt#L166
            // Envoi des références
            println!("Upload pack command detected");
            // Note: la commande peut aussi contenir des paramètres comme version=1.
            // Pour cet article, on ignore ce cas.
            self.send_references_capabilities();
        } else if pkt.find(WANT_CMD) == Some(4) {
```

Contenu masqué

```
80              // Référence :
81              //
    https://github.com/git/git/blob/master/Documentation/technical/pack-
    protocol.txt#L229
82              // NOTE: Un client peut envoyer plusieurs want. À noter
    aussi que le premier want est suivi des
83              // options que le client souhaite. Pour simplifier, ces
    cas sont ignorés ici.
84              self.wanted = String::from(pkt.get(9..49).unwrap()); //
    on prend juste le commit id
85              println!("Detected wanted commit: {}", self.wanted);
86          } else if pkt.find(HAVE_CMD) == Some(4) {
87              // Référence :
88              //
    https://github.com/git/git/blob/master/Documentation/technical/pack-
    protocol.txt#L390
89              // NOTE: possible d'améliorer cette partie pour le
    multi-ack
90              let have_commit =
    String::from(pkt.get(9..49).unwrap()); // on prend juste le
    commit id
91              if self.common.is_empty() {
92                  if
    self.repository.find_commit(Oid::from_str(&*have_commit).unwrap()).is_ok()
    {
93                      self.common = have_commit.clone();
94                      println!("Set common commit to: {}",
    self.common);
95                  }
96              }
97              self.have.push(have_commit);
98          } else if pkt == DONE_PKT {
99              // Référence :
100             //
    https://github.com/git/git/blob/master/Documentation/technical/pack-
    protocol.txt#L390
101             // NOTE: multi-ack ignoré ici. Si pas de base commune,
    on envoie seulement NAK.
102
    println!("Peer negotiation is done. Answering to want order");
103             let send_data = match self.common.is_empty() {
104                 true => self.nak(),
105                 false => self.ack_first(),
106             };
107             if send_data {
108                 self.send_pack_data();
109             }
110             self.stop = true;
111         } else if pkt == FLUSH_PKT {
112             if !self.have.is_empty() {
```

Contenu masqué

```
113                 // Référence :
114                 //
    https://github.com/git/git/blob/master/Documentation/technical/pack-
    protocol.txt#L390
115                 self.ack_common();
116                 self.nak();
117             }
118         } else {
119             println!("Unwanted packet received: {}", pkt);
120         }
121         self.buf.len() != 0
122     }
123
124     fn send_references_capabilities(&self) {
125         let current_head =
    self.repository.refname_to_id("HEAD").unwrap();
126         let mut capabilities =
    format!("{} HEAD\0side-band side-band-64k shallow no-progress include-tag"
    current_head);
127         capabilities = format!("{:04x}{}\n", capabilities.len() + 5
    /* taille + \n */, capabilities);
128
129         for name in self.repository.referencess().unwrap().names()
    {
130             let reference : &str = name.unwrap();
131             let oid =
    self.repository.refname_to_id(reference).unwrap();
132             capabilities += &*format!("{:04x}{} {}\n", 6 /* taille
    + espace + \n */ + 40 /* oid */ + reference.len(), oid,
    reference);
133         }
134
135         print!("Send: {}", capabilities);
136
    self.channel.lock().unwrap().send(capabilities.as_bytes().to_vec()).unwrap
137         println!("Send: {}", FLUSH_PKT);
138
    self.channel.lock().unwrap().send(FLUSH_PKT.as_bytes().to_vec()).unwrap();
139     }
140
141     fn nak(&self) -> bool {
142         print!("Send: {}", NAK_PKT);
143
    self.channel.lock().unwrap().send(NAK_PKT.as_bytes().to_vec()).is_ok()
144     }
145
146     fn ack_common(&self) -> bool {
147         let length = 18 /* taille + ACK + espace * 2 + continue +
    \n */ + self.common.len();
```

Contenu masqué

```
148         let msg = format!("{:04x}ACK {} continue\n", length,
    self.common);
149         print!("Send: {}", msg);
150 
    self.channel.lock().unwrap().send(msg.as_bytes().to_vec()).is_ok()
151     }
152 
153     fn ack_first(&self) -> bool {
154         let length = 9 /* taille + ACK + espace + \n */ +
    self.common.len();
155         let msg = format!("{:04x}ACK {}\n", length, self.common);
156         print!("Send: {}", msg);
157 
    self.channel.lock().unwrap().send(msg.as_bytes().to_vec()).is_ok()
158     }
159 
160     fn send_pack_data(&self) {
161         println!("Send: [PACKFILE]");
162         // Note: Ici, on ajoute tous les commits tant qu'on à pas
163         // trouvé un commit annoncé par le client et qu'on ne sois
164         // pas sûr d'avoir parcouru toutes les branches (ou
    jusqu'au
165         // commit initial).
166         let mut pb = self.repository.packbuilder().unwrap();
167         let fetched = Oid::from_str(&*self.wanted).unwrap();
168         let mut revwalk = self.repository.revwalk().unwrap();
169         let _ = revwalk.push(fetched);
170         let _ = revwalk.set_sorting(Sort::TOPOLOGICAL);
171 
172         let mut parents : Vec<String> = Vec::new();
173         let mut have = false;
174 
175         while let Some(oid) = revwalk.next() {
176             let oid = oid.unwrap();
177             let oid_str = oid.to_string();
178             have |= self.have.iter().find(|&o| *o ==
    oid_str).is_some();
179             if let Some(pos) = parents.iter().position(|p| *p ==
    oid_str) {
180                 parents.remove(pos);
181             }
182             if have && parents.is_empty() {
183                 // Commit initial
184                 break;
185             }
186             let _ = pb.insert_commit(oid);
187             let commit = self.repository.find_commit(oid).unwrap();
188             let mut commit_parents = commit.parents();
189             // On doit être sûr de parcourir tous les chemins
190             while let Some(p) = commit_parents.next() {
```

Contenu masqué

```
191                 parents.push(p.id().to_string());
192             }
193         }
194 
195         // Note: Packbuilder possède des fonctions pour écrire des chunks et pas tout le packfile d'un coup.
196         let mut data = Buf::new();
197         let _ = pb.write_buf(&mut data);
198 
199         let len = data.len();
200         let data : Vec<u8> = data.to_vec();
201         let mut sent = 0;
202         while sent < len {
203             // cf https://github.com/git/git/blob/master/Documentation/technical/pack-protocol.txt#L166
204             // Si side-band-64k est spécifié, on doit envoyer jusqu'à 65519 octets + 1 de contrôle par pkt-line.
205             let pkt_size = min(65515, len - sent);
206             // Le paquet: Size (4 octets), Control byte (0x01 pour les données), pack data.
207             let pkt = format!("{:04x}", pkt_size + 5 /* taille + control */);
208 
    self.channel.lock().unwrap().send(pkt.as_bytes().to_vec()).unwrap();
209 
    self.channel.lock().unwrap().send(b"\x01".to_vec()).unwrap();
210 
    self.channel.lock().unwrap().send(data[sent..(sent+pkt_size)].to_vec()).unw
211             sent += pkt_size;
212         }
213 
214         println!("Send: {}", FLUSH_PKT);
215         // Et on fini par un FLUSH
216 
    self.channel.lock().unwrap().send(FLUSH_PKT.as_bytes().to_vec()).unwrap();
217     }
218 }
```

Retourner au texte.

Contenu masqué n°3

```rust
mod wolftransport;
mod server;

use bichannel::channel;
use wolftransport::WolfChannel;
use git2::build::RepoBuilder;
use server::Server;
use std::env;
use std::sync::{Arc, Mutex};
use std::thread;
use std::path::Path;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() != 3 {
        println!("Usage: ./p2p-internal-git <src_dir> <dest_dir>");
        return;
    }

    let src_dir = args[1].clone();
    let dest_dir = Path::new(&args[2]);

    if dest_dir.is_dir() {
        println!("Can't clone into an existing directory");
        return;
    }

    // On utilise ici bichannel::channel() à des fins de démos
    // mais on peut le remplacer par le transport que l'on souhaite
    // par exemple un transport TLS.
    let (server_channel, transport_channel) = channel();
    let transport_channel = Arc::new(Mutex::new(WolfChannel {
        channel: transport_channel
    }));
    let server_channel = Arc::new(Mutex::new(WolfChannel {
        channel: server_channel
    }));
    unsafe {
        wolftransport::register(transport_channel);
    }

    let server = thread::spawn(move || {
        println!("Starting server for {}", src_dir);
        let mut server = Server::new(server_channel, &*src_dir);
        server.run();
    });

```

Contenu masqué

```
48      // Note: "wolf://" est pour utiliser notre transport.
        localhost/zds n'est pas utilisé
49      // parce que notre serveur ne gère qu'un seul répertoire.
50      RepoBuilder::new().clone("wolf://localhost/zds",
        dest_dir).unwrap();
51      println!("Cloned into {:?}!", dest_dir);
52
53      server.join().expect("The server panicked");
54  }
```

et notre `Cargo.toml`:

```
1  [package]
2  name = "p2p-internals-git"
3  version = "0.1.0"
4  authors = ["AmarOk"]
5  edition = "2018"
6
7  [dependencies]
8  bichannel = "0.0.4"
9  git2 = { git = "https://github.com/AmarOk1412/git2-rs" }
```

Retourner au texte.