Bitcoin Core

La référence d'implémentation

2022-03-04

# Bitcoin Core : La référence d'implémentation

Bitcoin est un projet *source libre* et le code source est disponible sous une licence ouverte (MIT), téléchargeable et utilisable à n'importe quelle fin. Source libre signifie plus que simplement libre d'utilisation. Cela signifie également que le bitcoin est développé par une communauté ouverte de bénévoles. Au début, cette communauté se composait uniquement de Satoshi Nakamoto. En 2016, le code source de Bitcoin comptait plus de 400 contributeurs avec une douzaine de développeurs travaillant sur le code presque à temps plein et plusieurs dizaines d'autres à temps partiel. Tout le monde peut contribuer au code, y compris vous !

("Bitcoin Core", "implémentation de référence")Lorsque bitcoin a été créé par Satoshi Nakamoto, le logiciel était en fait terminé avant le livre blanc reproduit dans [???](#satoshi_whitepaper). Satoshi voulait s'assurer que cela fonctionnait avant d'écrire à ce sujet. Cette première implémentation, alors simplement connue sous le nom de "Bitcoin" ou "client Satoshi", a été fortement modifiée et améliorée. Il a évolué vers ce qu'on appelle *Bitcoin Core*, pour le différencier des autres implémentations compatibles. Bitcoin Core est la *mise en œuvre de référence* du système Bitcoin, ce qui signifie qu'il s'agit de la référence faisant autorité sur la manière dont chaque partie de la technologie doit être mise en œuvre. Bitcoin Core implémente tous les aspects du bitcoin, y compris les portefeuilles, un moteur de validation des transactions et des blocs et un nœud de réseau complet dans le réseau Bitcoin pair à pair.

Même si Bitcoin Core inclut une implémentation de référence d'un portefeuille, celui-ci n'est pas destiné à être utilisé comme portefeuille de production pour les utilisateurs ou pour les applications. Il est conseillé aux développeurs d'applications de créer des portefeuilles en utilisant des normes modernes telles que BIP-39 et BIP-32 (voir [???](#mnemonic_code_words) et [???](#hd_wallets)). BIP signifie *Bitcoin Improvement Proposal*.

[variablelist\_title](#bitcoin_core_architecture) montre l'architecture de Bitcoin Core.

image

images/mbc2\_0301.png["Architecture de base Bitcoin"]

## Environnement de développement Bitcoin

Si vous êtes un développeur, vous souhaiterez configurer un environnement de développement avec tous les outils, bibliothèques et logiciels de support pour le développement d’applications bitcoin. Dans ce chapitre très technique, nous allons parcourir ce processus étape par étape. Si le matériel devient trop dense (et que vous n'êtes pas en train de mettre en place un environnement de développement), n'hésitez pas à passer au chapitre suivant, qui est moins technique.

## Compilation de Bitcoin Core à partir du code source

Le code source de Bitcoin Core peut être téléchargé sous forme d'archive ou en clonant le référentiel source faisant autorité à partir de GitHub. Sur la [page de téléchargement de Bitcoin Core](https://bitcoincore.org/bin/), sélectionnez la version la plus récente et téléchargez l'archive compressée du code source, par exemple, bitcoin-0.15.0.2.tar.gz. Vous pouvez également utiliser la ligne de commande git pour créer une copie locale du code source à partir de la [page GitHub de bitcoin](https://github.com/bitcoin/bitcoin).

Dans de nombreux exemples de ce chapitre, nous utiliserons l'interface de de commande du système d'exploitation ( également appelé "shell" ou "interface système" ou "coque logicielle"), accessible via une application "terminal". L’interface système affichera une invite; vous tapez une commande; et l’interface système répond avec du texte et une nouvelle invite pour votre prochaine commande. L'invite peut sembler différente sur votre système, mais dans les exemples suivants, elle est indiquée par un symbole $. Dans les exemples, lorsque vous voyez du texte après un symbole $, ne tapez pas le symbole $ mais tapez la commande qui le suit immédiatement, puis appuyez sur Entrée pour exécuter la commande. Dans les exemples, les lignes sous chaque commande sont les réponses du système d'exploitation à cette commande. Lorsque vous verrez le prochain préfixe $, vous saurez qu'il s'agit d'une nouvelle commande et vous devrez répéter le processus.

Dans cet exemple, nous utilisons la commande git pour créer une copie locale (un "clone") du code source :

$ git clone https://github.com/bitcoin/bitcoin.git  
Cloning into 'bitcoin'...  
remote: Counting objects: 102071, done.  
remote: Compressing objects: 100% (10/10), done.  
Receiving objects: 100% (102071/102071), 86.38 MiB | 730.00 KiB/s, done.  
remote: Total 102071 (delta 4), reused 5 (delta 1), pack-reused 102060  
Resolving deltas: 100% (76168/76168), done.  
Checking connectivity... done.  
$

Git est le système de contrôle de version distribué le plus largement utilisé et un élément essentiel de la boîte à outils de tout développeur de logiciels. Vous devrez peut-être installer la commande git, ou une interface utilisateur graphique pour git, sur votre système d'exploitation si vous ne l'avez pas déjà.

Une fois l'opération de clonage git terminée, vous aurez une copie locale complète du référentiel de code source dans le répertoire *bitcoin*. Accédez à ce répertoire en tapant \*\*cd bitcoin\*\* à l'invite :

$ cd bitcoin

### Sélection d'une version de Bitcoin Core

Par défaut, la copie locale sera synchronisée avec le code le plus récent, qui peut être une version instable ou bêta de bitcoin. Avant de compiler le code, sélectionnez une version spécifique en consultant une version avec la commande Git *tag*. Cela synchronisera la copie locale avec un instantané (snapshot) spécifique du référentiel de code identifié par une balise de mot-clé (le *tag*.) Les balises sont utilisées par les développeurs pour marquer des versions spécifiques du code par numéro de version. Tout d'abord, pour trouver les balises disponibles, nous utilisons la commande git tag :

$ git tag  
v0.1.5  
v0.1.6test1  
v0.10.0  
...  
v0.11.2  
v0.11.2rc1  
v0.12.0rc1  
v0.12.0rc2  
...

La liste des balises montre toutes les versions publiées de bitcoin. Par convention, les *release candidates* (ou *versions d’évaluations*), qui sont destinées aux tests, ont le suffixe "rc". Les versions stables pouvant être exécutées sur des systèmes de production n'ont pas de suffixe. Dans la liste précédente, sélectionnez la version la plus élevée, qui au moment de la rédaction était v0.15.0. Pour synchroniser le code local avec cette version, utilisez la commande git checkout :

$ git checkout v0.15.0  
HEAD is now at 3751912... Merge #11295: doc: Old fee\_estimates.dat are discarded by 0.15.0

Vous pouvez confirmer que vous avez la version souhaitée "extraite" en lançant la commande git status :

$ git status  
HEAD detached at v0.15.0  
nothing to commit, working directory clean

### Configuration de la construction de Bitcoin Core

Le code source inclut la documentation, qui peut être trouvée dans un certain nombre de fichiers. Passez en revue la documentation principale située dans *README.md* dans le répertoire *bitcoin* en tapant \*\*more README.md\*\* à l'invite et en utilisant la barre d'espace pour passer à la page suivante. Dans ce chapitre, nous allons construire le client Bitcoin en ligne de commande, également connu sous le nom de bitcoind sous Linux. Passez en revue les instructions pour compiler le client de ligne de commande bitcoind sur votre plate-forme en tapant \*\*more doc/build-unix.md\*\*. Des instructions alternatives pour macOS et Windows peuvent être trouvées dans le répertoire *doc*, comme *build-osx.md* ou *build-windows.md*, respectivement.

Examinez attentivement les prérequis de construction, qui se trouvent dans la première partie de la documentation de construction. Ce sont des bibliothèques qui doivent être présentes sur votre système avant de pouvoir commencer à compiler Bitcoin. Si ces conditions préalables sont manquantes, le processus de génération échouera avec une erreur. Si cela se produit parce que vous avez manqué une condition préalable, vous pouvez l'installer, puis reprendre le processus de génération là où vous l'avez laissé. En supposant que les prérequis sont installés, vous démarrez le processus de génération en générant un ensemble de scripts de génération à l'aide du script *autogen.sh*.

$ ./autogen.sh  
...  
glibtoolize: copying file 'build-aux/m4/libtool.m4'  
glibtoolize: copying file 'build-aux/m4/ltoptions.m4'  
glibtoolize: copying file 'build-aux/m4/ltsugar.m4'  
glibtoolize: copying file 'build-aux/m4/ltversion.m4'  
...  
configure.ac:10: installing 'build-aux/compile'  
configure.ac:5: installing 'build-aux/config.guess'  
configure.ac:5: installing 'build-aux/config.sub'  
configure.ac:9: installing 'build-aux/install-sh'  
configure.ac:9: installing 'build-aux/missing'  
Makefile.am: installing 'build-aux/depcomp'  
...

Le script *autogen.sh* crée un ensemble de scripts de configuration automatique qui interrogeront votre système pour découvrir les paramètres corrects et s'assurer que vous disposez de toutes les bibliothèques nécessaires pour compiler le code. Le plus important d'entre eux est le script configure qui offre un certain nombre d'options différentes pour personnaliser le processus de construction. Tapez \*\*./configure --help\*\* pour voir les différentes options:

$ ./configure --help  
`configure' configures Bitcoin Core 0.15.0 to adapt to many kinds of systems.  
  
Usage: ./configure [OPTION]... [VAR=VALUE]...  
  
...  
Optional Features:  
 --disable-option-checking ignore unrecognized --enable/--with options  
 --disable-FEATURE do not include FEATURE (same as --enable-FEATURE=no)  
 --enable-FEATURE[=ARG] include FEATURE [ARG=yes]  
  
 --enable-wallet enable wallet (default is yes)  
  
 --with-gui[=no|qt4|qt5|auto]  
...

Le script configure vous permet d'activer ou de désactiver certaines fonctionnalités de bitcoind grâce à l'utilisation des drapeaux --enable-FEATURE et --disable-FEATURE, où FEATURE est remplacé par le nom de la fonctionnalité, comme indiqué dans la sortie d'aide. Dans ce chapitre, nous allons construire le client bitcoind avec toutes les fonctionnalités par défaut. Nous n'utiliserons pas les indicateurs de configuration, mais vous devriez les consulter pour comprendre quelles fonctionnalités facultatives font partie du client. Si vous êtes dans un environnement universitaire, les restrictions du laboratoire informatique peuvent vous obliger à installer des applications dans votre répertoire personnel (par exemple, en utilisant --prefix=$HOME).

Voici quelques options utiles qui remplacent le comportement par défaut du script configure:

--prefix=$HOME

Cela remplace l'emplacement d'installation par défaut (qui est

/usr/local/

) pour l'exécutable résultant. Utilisez

$HOME

pour tout mettre dans votre répertoire personnel, ou un chemin différent.

--disable-wallet

Ceci est utilisé pour désactiver l'implémentation du portefeuille de référence.

--with-incompatible-bdb

Si vous construisez un portefeuille, autorisez l'utilisation d'une version incompatible de la bibliothèque Berkeley DB.

--with-gui=no

Ne construisez pas l'interface utilisateur graphique, qui nécessite la bibliothèque Qt. Cela construit uniquement le bitcoin du serveur et de la ligne de commande.

Ensuite, exécutez le script configure pour découvrir automatiquement toutes les bibliothèques nécessaires et créer un script de construction personnalisé pour votre système :

$ ./configure  
checking build system type... x86\_64-unknown-linux-gnu  
checking host system type... x86\_64-unknown-linux-gnu  
checking for a BSD-compatible install... /usr/bin/install -c  
checking whether build environment is sane... yes  
checking for a thread-safe mkdir -p... /bin/mkdir -p  
checking for gawk... gawk  
checking whether make sets $(MAKE)... yes  
...  
[de nombreuses pages de tests de configuration suivent]  
...  
$

Si tout s'est bien passé, la commande configure se terminera par la création des scripts de construction personnalisés qui nous permettront de compiler bitcoind. S'il y a des bibliothèques manquantes ou des erreurs, la commande configure se terminera par une erreur au lieu de créer les scripts de construction. Si une erreur se produit, elle est probablement due à une bibliothèque manquante ou incompatible. Relisez à nouveau la documentation de build et assurez-vous d'installer les prérequis manquants. Ensuite, exécutez à nouveau configure et voyez si cela corrige l'erreur.

### Construire les exécutables Bitcoin Core

Ensuite, vous allez compiler le code source, un processus qui peut prendre jusqu'à une heure, en fonction de la vitesse de votre processeur et de la mémoire disponible. Pendant le processus de compilation, vous devriez voir une sortie toutes les quelques secondes ou toutes les quelques minutes, ou une erreur si quelque chose ne va pas. Si une erreur se produit ou si le processus de compilation est interrompu, il peut être repris à tout moment en tapant à nouveau make. Tapez \*\*make\*\* pour lancer la compilation de l'application exécutable :

$ make  
Making all in src  
 CXX crypto/libbitcoinconsensus\_la-hmac\_sha512.lo  
 CXX crypto/libbitcoinconsensus\_la-ripemd160.lo  
 CXX crypto/libbitcoinconsensus\_la-sha1.lo  
 CXX crypto/libbitcoinconsensus\_la-sha256.lo  
 CXX crypto/libbitcoinconsensus\_la-sha512.lo  
 CXX libbitcoinconsensus\_la-hash.lo  
 CXX primitives/libbitcoinconsensus\_la-transaction.lo  
 CXX libbitcoinconsensus\_la-pubkey.lo  
 CXX script/libbitcoinconsensus\_la-bitcoinconsensus.lo  
 CXX script/libbitcoinconsensus\_la-interpreter.lo  
  
[... de nombreux autres messages de compilation suivent ...]  
  
$

Sur un système rapide avec plusieurs processeurs, vous souhaiterez peut-être définir le nombre de tâches de compilation parallèles. Par exemple, make -j 2 utilisera deux cœurs s'ils sont disponibles. Si tout se passe bien, Bitcoin Core est maintenant compilé. Vous devez exécuter la suite de tests unitaires avec make check pour vous assurer que les bibliothèques liées ne sont pas cassées de manière évidente. La dernière étape consiste à installer les différents exécutables sur votre système à l'aide de la commande make install. Vous serez peut-être invité à saisir votre mot de passe utilisateur, car cette étape nécessite des privilèges d'administrateur:

$ make check && sudo make install  
Password:  
Making install in src  
 ../build-aux/install-sh -c -d '/usr/local/lib'  
libtool: install: /usr/bin/install -c bitcoind /usr/local/bin/bitcoind  
libtool: install: /usr/bin/install -c bitcoin-cli /usr/local/bin/bitcoin-cli  
libtool: install: /usr/bin/install -c bitcoin-tx /usr/local/bin/bitcoin-tx  
...  
$

L'installation par défaut de bitcoind le place dans */usr/local/bin*. Vous pouvez confirmer que Bitcoin Core est correctement installé en demandant au système le chemin des exécutables, comme suit:

$ which bitcoind  
/usr/local/bin/bitcoind  
  
$ which bitcoin-cli  
/usr/local/bin/bitcoin-cli

## Exécution d'un nœud Bitcoin Core

Le réseau pair à pair de Bitcoin est composé de "nœuds" de réseau, gérés principalement par des bénévoles et certaines des entreprises qui créent des applications bitcoin. Ceux qui exécutent des nœuds Bitcoin ont une vue directe et faisant autorité sur la chaîne de blocs Bitcoin, avec une copie locale de toutes les transactions, validées indépendamment par leur propre système. En exécutant un nœud, vous n'avez pas besoin de faire appel à un tiers pour valider une transaction. De plus, en exécutant un nœud Bitcoin, vous contribuez au réseau Bitcoin en le rendant plus robuste et sécuritaire.

Cependant, l'exécution d'un nœud nécessite un système connecté en permanence avec suffisamment de ressources pour traiter toutes les transactions bitcoin. Selon que vous choisissez d'indexer toutes les transactions et de conserver une copie complète de la chaîne de blocs, vous pouvez également avoir besoin de beaucoup d'espace disque et de mémoire RAM. Au début de 2021, un nœud à index complet a besoin de 2 Go de RAM et d'un minimum de 360 Go d'espace disque (voir <https://www.blockchain.com/charts/blocks-size>). Les nœuds Bitcoin transmettent et reçoivent également des transactions et des blocs Bitcoin, consommant de la bande passante Internet. Si votre connexion Internet est limitée, a un faible plafond de données ou est mesurée (facturée au gigabit), vous ne devriez probablement pas exécuter un nœud Bitcoin, ou l'exécuter d'une manière qui limite sa bande passante (voir [example\_title](#constrained_resources)).

Bitcoin Core conserve une copie complète de la chaîne de blocs par défaut, avec chaque transaction qui s'est déjà produite sur le Bitcoin réseau depuis sa création en 2009. Cet ensemble de données a une taille de dizaines de gigaoctets et est téléchargé de manière incrémentielle sur plusieurs jours ou semaines, en fonction de la vitesse de votre CPU et de votre connexion Internet. Bitcoin Core ne pourra pas traiter les transactions ou mettre à jour les soldes des comptes tant que l'ensemble de données complet de la chaîne de blocs n'aura pas été téléchargé. Assurez-vous que vous disposez de suffisamment d'espace disque, de bande passante et de temps pour effectuer la synchronisation initiale. Vous pouvez configurer Bitcoin Core pour réduire la taille de la chaîne de blocs en supprimant les anciens blocs (voir [example\_title](#constrained_resources)), mais il téléchargera tout de même l'intégralité de l'ensemble de données avant de les supprimer.

Malgré ces besoins en ressources, des milliers de bénévoles gèrent des nœuds Bitcoin. Certains fonctionnent sur des systèmes aussi simples qu'un Raspberry Pi (un ordinateur à 35$ US de la taille d'un paquet de cartes). De nombreux volontaires exécutent également des nœuds Bitcoin sur des serveurs loués, généralement une variante de Linux. Une instance de *Virtual Private Server* (VPS ou serveur virtuel privé) ou *Cloud Computing Server* (ou serveur en infonuagique) peut être utilisée pour exécuter un nœud Bitcoin. Ces serveurs peuvent être loués pour 25$ à 50$ US par mois auprès de divers fournisseurs.

Pourquoi voudriez-vous exécuter un nœud ? Voici quelques-unes des raisons les plus courantes :

* Si vous développez un logiciel bitcoin et que vous devez vous fier à un nœud Bitcoin pour un accès programmable (API) au réseau et à la blockchain.
* Si vous créez des applications qui doivent valider les transactions selon les règles de consensus de bitcoin. En règle générale, les éditeurs de logiciels Bitcoin exécutent plusieurs nœuds.
* Si vous souhaitez prendre en charge le bitcoin. L'exécution d'un nœud rend le réseau plus robuste et capable de servir plus de portefeuilles, plus d'utilisateurs et plus de transactions.
* Si vous ne souhaitez pas vous fier à un tiers pour traiter ou valider vos transactions.

Si vous lisez ce livre et êtes intéressé par le développement d'un logiciel Bitcoin, vous devriez exécuter votre propre nœud.

### Configuration du nœud Bitcoin Core

( "sécurité", "mots de passe")Bitcoin Core recherchera un fichier de configuration dans son répertoire de données à chaque démarrage. Dans cette section, nous examinerons les différentes options de configuration et créerons un fichier de configuration. Pour localiser le fichier de configuration, exécutez bitcoind -printtoconsole dans votre terminal et recherchez les deux premières lignes.

$ bitcoind -printtoconsole  
Bitcoin version v0.15.0  
Using the 'standard' SHA256 implementation  
Using data directory /home/ubuntu/.bitcoin/  
Using config file /home/ubuntu/.bitcoin/bitcoin.conf  
...  
[beaucoup plus de sortie de débogage]  
...

Vous pouvez appuyer sur Ctrl-C pour fermer le nœud une fois que vous avez déterminé l'emplacement du fichier de configuration. Habituellement, le fichier de configuration se trouve dans le répertoire de données *.bitcoin* sous le répertoire personnel de votre utilisateur. Il n'est pas créé automatiquement, mais vous pouvez créer un fichier de configuration de démarrage en copiant et collant à partir de l'exemple [example\_title](#full_index_node) ci-dessous. Vous pouvez créer ou modifier le fichier de configuration dans votre éditeur préféré.

Bitcoin Core propose plus de 100 options de configuration qui modifient le comportement du nœud de réseau, le stockage de la chaîne de blocs et de nombreux autres aspects de son fonctionnement. Pour voir une liste de ces options, exécutez bitcoind --help :

$ bitcoind --help  
Bitcoin Core Daemon version v0.15.0  
  
Usage:  
 bitcoind [options] Start Bitcoin Core Daemon  
  
Options:  
  
 -?  
 Print this help message and exit  
  
 -version  
 Print version and exit  
  
 -alertnotify=<cmd>  
 Execute command when a relevant alert is received or we see a really  
 long fork (%s in cmd is replaced by message)  
...  
[beaucoup plus d&#39;options]  
...  
  
 -rpcthreads=<n>  
 Set the number of threads to service RPC calls (default: 4)

Voici quelques-unes des options les plus importantes que vous pouvez définir dans le fichier de configuration, ou en tant que paramètres de ligne de commande pour bitcoind :

alertnotify

Exécute une commande ou un script spécifié pour envoyer des alertes d'urgence au propriétaire de ce nœud, généralement par courriel.

conf

Un emplacement alternatif pour le fichier de configuration. Cela n'a de sens qu'en tant que paramètre de ligne de commande pour bitcoind, car il ne peut pas se trouver dans le fichier de configuration auquel il se réfère.

datadir

Sélectionnez le répertoire et le système de fichiers dans lesquels vous placer toutes les données de la chaîne de blocs. Par défaut, il s'agit d’un sous-répertoire de *.bitcoin* de votre répertoire personnel. Assurez-vous que ce système de fichiers dispose de plusieurs gigaoctets d'espace libre.

prune

Réduisez les besoins en espace disque à ce nombre de mégaoctets, en supprimant les anciens blocs. Utilisez ceci sur un nœud à ressources limitées qui ne peut pas contenir la chaîne de blocs complète.

txindex

Maintient un index de toutes les transactions. Cela signifie une copie complète de la chaîne de blocs. Cela vous permet de récupérer par programmation toute transaction par identifiant (ID).

dbcache

La taille du cache UTXO. La valeur par défaut est 450 Mio. Augmentez cette valeur sur le matériel haut de gamme et réduisez la taille sur le matériel bas de gamme pour économiser de la mémoire au détriment des entrées-sorties des disques lents.

maxconnections

Définissez le nombre maximal de nœuds à partir desquels accepter les connexions. Le réduire par défaut réduira votre consommation de bande passante. À utiliser si vous avez un plafond de données ou si vous payez au gigaoctet.

maxmempool

Limite le bassin de mémoire de transaction à ce nombre de mégaoctets. Utilisez-le pour réduire l'utilisation de la mémoire sur les nœuds à mémoire limitée.

maxreceivebuffer/maxsendbuffer

Limite la mémoire tampon par connexion à ce nombre de multiples de 1000 octets. À utiliser sur les nœuds à mémoire limitée.

minrelaytxfee

Définissez le taux de frais minimum pour la transaction que vous relayerez. En dessous de cette valeur, la transaction est traitée non standard, rejetée du bassin de transactions et non relayée.

Par défaut, Bitcoin Core construit une base de données contenant *uniquement* les transactions liées au portefeuille de l'utilisateur. Si vous voulez pouvoir accéder à *n'importe quelle* transaction avec des commandes comme getrawtransaction (voir [Explorer et décoder les transactions](#exploring_and_decoding_transanctions)), vous devez configurer Bitcoin Core pour créer un index de transaction complet, ce qui peut être réalisé avec l'option txindex. Définissez txindex=1 dans le fichier de configuration de Bitcoin Core. Si vous ne définissez pas cette option au début et que vous la réglez ensuite sur l'indexation complète, vous devez redémarrer bitcoind avec l'option -reindex et attendre qu'elle reconstruise l'index.

[example\_title](#full_index_node) montre comment vous pouvez combiner les options précédentes, avec un nœud entièrement indexé, fonctionnant comme un backend API pour une application bitcoin.

alertnotify=myemailscript.sh &quot;Alerte : %s&quot;  
datadir=/lotsofspace/bitcoin  
index\_tx=1

[example\_title](#constrained_resources) montre un nœud à ressources limitées s'exécutant sur un serveur plus petit.

alertnotify=myemailscript.sh &quot;Alerte : %s&quot;  
maxconnexions=15  
prune=5000  
dbcache=150  
maxmempool=150  
maxreceivebuffer=2500  
maxsendbuffer=500

Une fois que vous avez modifié le fichier de configuration et défini les options qui représentent le mieux vos besoins, vous pouvez tester bitcoind avec cette configuration. Exécutez Bitcoin Core avec l'option printtoconsole pour s'exécuter au premier plan avec une sortie vers la console :

$ bitcoind -printtoconsole  
  
Bitcoin version v0.15.0  
InitParameterInteraction: parameter interaction: -whitelistforcerelay=1 -> setting -whitelistrelay=1  
Assuming ancestors of block 0000000000000000003b9ce759c2a087d52abc4266f8f4ebd6d768b89defa50a have valid signatures.  
Using the 'standard' SHA256 implementation  
Default data directory /home/ubuntu/.bitcoin  
Using data directory /lotsofspace/.bitcoin  
Using config file /home/ubuntu/.bitcoin/bitcoin.conf  
Using at most 125 automatic connections (1048576 file descriptors available)  
Using 16 MiB out of 32/2 requested for signature cache, able to store 524288 elements  
Using 16 MiB out of 32/2 requested for script execution cache, able to store 524288 elements  
Using 2 threads for script verification  
HTTP: creating work queue of depth 16  
No rpcpassword set - using random cookie authentication  
Generated RPC authentication cookie /lotsofspace/.bitcoin/.cookie  
HTTP: starting 4 worker threads  
init message: Verifying wallet(s)...  
Using BerkeleyDB version Berkeley DB 4.8.30: (April 9, 2010)  
Using wallet wallet.dat  
CDBEnv::Open: LogDir=/lotsofspace/.bitcoin/database ErrorFile=/lotsofspace/.bitcoin/db.log  
scheduler thread start  
Cache configuration:  
\* Using 250.0MiB for block index database  
\* Using 8.0MiB for chain state database  
\* Using 1742.0MiB for in-memory UTXO set (plus up to 286.1MiB of unused mempool space)  
init message: Loading block index...  
Opening LevelDB in /lotsofspace/.bitcoin/blocks/index  
Opened LevelDB successfully  
  
[... plus de messages de démarrage ...]

Vous pouvez appuyer sur Ctrl-C pour interrompre le processus une fois que vous êtes convaincu qu'il charge les paramètres correctement et s'exécute comme prévu.

Pour exécuter Bitcoin Core en arrière-plan en tant que processus, démarrez-le avec l'option daemon, en tant que bitcoind -daemon.

Pour surveiller la progression et l'état d'exécution de votre nœud Bitcoin, utilisez la commande bitcoin-cli getblockchaininfo :

$ bitcoin-cli getblockchaininfo

{  
 "chain": "main",  
 "blocks": 0,  
 "headers": 83999,  
 "bestblockhash": "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",  
 "difficulty": 1,  
 "mediantime": 1231006505,  
 "verificationprogress": 3.783041623201835e-09,  
 "chainwork": "0000000000000000000000000000000000000000000000000000000100010001",  
 "pruned": false,  
 [...]  
}

Cela montre un nœud avec une hauteur de blockchain de 0 blocs et 83999 en-têtes. Le nœud récupère actuellement les en-têtes de bloc de la meilleure chaîne et continue ensuite à télécharger les blocs complets.

Une fois que vous êtes satisfait des options de configuration que vous avez sélectionnées, vous devez ajouter *bitcoin* aux scripts de démarrage de votre système d'exploitation, afin qu'il s'exécute en continu et au redémarrage du système d'exploitation. Vous trouverez un certain nombre d'exemples de scripts de démarrage pour différents systèmes d'exploitation dans le répertoire source de bitcoin sous *contrib/init* et un fichier *README.md* indiquant quel système utilise quel script.("", startref="BNcore03")

## Interface de programmation d'application (API) de base Bitcoin

Le client Bitcoin Core implémente une interface JSON-RPC accessible également à l'aide de l'assistant de ligne de commande bitcoin-cli. La ligne de commande nous permet d'expérimenter de manière interactive les fonctionnalités qui sont également disponibles par programmation via l'API. Pour commencer, appelez la commande help pour voir une liste des commandes RPC bitcoin disponibles :

$ bitcoin-cli help  
addmultisigaddress nrequired ["key",...] ( "account" )  
addnode "node" "add|remove|onetry"  
backupwallet "destination"  
createmultisig nrequired ["key",...]  
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}  
decoderawtransaction "hexstring"  
...  
...  
verifymessage "bitcoinaddress" "signature" "message"  
walletlock  
walletpassphrase "passphrase" timeout  
walletpassphrasechange "oldpassphrase" "newpassphrase"

Chacune de ces commandes peut prendre un certain nombre de paramètres. Pour obtenir une aide supplémentaire, une description détaillée et des informations sur les paramètres, ajoutez le nom de la commande après help. Par exemple, pour voir l'aide sur la commande getblockhash RPC :

$ bitcoin-cli help getblockhash  
getblockhash height  
  
Returns hash of block in best-block-chain at height provided.  
  
Arguments:  
1. height (numeric, required) The height index  
  
Result:  
"hash" (string) The block hash  
  
Examples:  
> bitcoin-cli getblockhash 1000  
> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getblockhash", "params": [1000] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/

À la fin des informations d'aide, vous verrez deux exemples de la commande RPC, utilisant l'assistant bitcoin-cli ou le client HTTP curl. Ces exemples montrent comment vous pouvez appeler la commande. Copiez le premier exemple et voyez le résultat :

$ bitcoin-cli getblockhash 1000  
00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09

Le résultat est un hachage de bloc, qui est décrit plus en détail dans les chapitres suivants. Mais pour l'instant, cette commande devrait renvoyer le même résultat sur votre système, démontrant que votre nœud Bitcoin Core est en cours d'exécution, accepte les commandes et dispose d'informations sur le bloc 1000 à vous renvoyer.

Dans les sections suivantes, nous allons démontrer quelques commandes RPC très utiles et leur sortie attendue.

### Obtenir des informations sur l'état du client Bitcoin Core

Bitcoin Core fournit des rapports d'état sur différents modules via l'interface JSON-RPC. Les commandes les plus importantes incluent getblockchaininfo, getmempoolinfo, getnetworkinfo et getwalletinfo.

La commande getblockchaininfo RPC de Bitcoin a été introduite plus tôt. La commande getnetworkinfo affiche des informations de base sur l'état du nœud de réseau Bitcoin. Utilisez bitcoin-cli pour l'exécuter :

$ bitcoin-cli getnetworkinfo

"version": 150000,  
 "subversion": "/Satoshi:0.15.0/",  
 "protocolversion": 70015,  
 "localservices": "000000000000000d",  
 "localrelay": true,  
 "timeoffset": 0,  
 "networkactive": true,  
 "connections": 8,  
 "networks": [  
 ...  
 detailed information about all networks (ipv4, ipv6 or onion)  
 ...  
 ],  
 "relayfee": 0.00001000,  
 "incrementalfee": 0.00001000,  
 "localaddresses": [  
 ],  
 "warnings": ""  
}

Les données sont renvoyées en JavaScript Object Notation (JSON), un format qui peut facilement être "consommé" par tous les langages de programmation mais qui est également assez lisible par l'homme. Parmi ces données, nous voyons les numéros de version du client logiciel bitcoin (150000) et du protocole Bitcoin (70015). On y voit le nombre actuel de connexions (8) et diverses informations sur le réseau Bitcoin et les paramètres liés à ce client.

Il faudra un certain temps, peut-être plus d'une journée, pour que le client bitcoind "rattrape" la hauteur actuelle de la chaîne de blocs lorsqu'il télécharge des blocs d'autres clients Bitcoin. Vous pouvez vérifier sa progression en utilisant getblockchaininfo pour voir le nombre de blocs connus.

### Explorer et décoder les transactions

Commandes : getrawtransaction, decoderawtransaction

Dans [???](#cup_of_coffee), Alice a acheté une tasse de café au Bob's Cafe. Sa transaction a été enregistrée sur la blockchain avec l'identifiant de transaction (txid) 0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2. Utilisons l'API pour récupérer et examiner cette transaction en transmettant l'identifiant de transaction en tant que paramètre :

$ bitcoin-cli getrawtransaction 0627052b6f28912f2703066a912ea577f2ce4da4caa5a↵ 5fbd8a57286c345c2f2 0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000↵ 000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4↵ ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813014↵ 10484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767↵ 89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adfffffffff0260e3160000000↵ 0001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9↵ 147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000

Un identifiant de transaction ne fait pas autorité tant qu'une transaction n'a pas été confirmée. L'absence d'un hachage de transaction dans la blockchain ne signifie pas que la transaction n'a pas été traitée. C'est ce qu'on appelle la "malléabilité des transactions", car les hachages de transaction peuvent être modifiés avant la confirmation dans un bloc. Après confirmation, le txid est immuable et fait autorité.

La commande getrawtransaction renvoie une transaction sérialisée en notation hexadécimale. Pour décoder cela, nous utilisons la commande decoderawtransaction, en passant les données hexadécimales en paramètre. Vous pouvez copier l'hex renvoyé par getrawtransaction et le coller en tant que paramètre dans decoderawtransaction :

$ bitcoin-cli decoderawtransaction 0100000001186f9f998a5aa6f048e51dd8419a14d8↵ a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba474↵ 6ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298↵ cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fd↵ e0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa↵ 336a8d752adfffffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5↵ d50f654e788acd0ef8000000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8↵ 88ac00000000

{ "txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2", "size": 258, "version": 1, "locktime": 0, "vin": [ { "txid": "7957a35fe64f80d234d76d83a2...8149a41d81de548f0a65a8a999f6f18", "vout": 0, "scriptSig": { "asm":"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1decc...", "hex":"483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1de..." }, "sequence": 4294967295 } ], "vout": [ { "value": 0.01500000, "n": 0, "scriptPubKey": { "asm": "OP\_DUP OP\_HASH160 ab68...5f654e7 OP\_EQUALVERIFY OP\_CHECKSIG", "hex": "76a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788ac", "reqSigs": 1, "type": "pubkeyhash", "addresses": [ "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA" ] } }, { "value": 0.08450000, "n": 1, "scriptPubKey": { "asm": "OP\_DUP OP\_HASH160 7f9b1a...025a8 OP\_EQUALVERIFY OP\_CHECKSIG", "hex": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac", "reqSigs": 1, "type": "pubkeyhash", "addresses": [ "1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK" ] } } ] }

Le décodage de la transaction montre tous les composants de cette transaction, y compris les entrées et les sorties de la transaction. Dans ce cas, nous voyons que la transaction qui a crédité notre nouvelle adresse de 15 millibits a utilisé une entrée et généré deux sorties. L'entrée de cette transaction était la sortie d'une transaction précédemment confirmée (indiquée par txid commençant par 7957a35fe). Les deux sorties correspondent au crédit de 15 millibits et une sortie avec retour à l'expéditeur.

Nous pouvons explorer davantage la chaîne de blocs en examinant la transaction précédente référencée par son txid dans cette transaction en utilisant les mêmes commandes (par exemple, getrawtransaction). En sautant d'une transaction à l'autre, nous pouvons suivre une chaîne de transactions à mesure que les pièces sont transmises d'adresse de propriétaire à adresse de propriétaire.

### Explorer les blocs

Commandes : getblock, getblockhash

L'exploration de blocs est similaire à l'exploration de transactions. Cependant, les blocs peuvent être référencés soit par la *hauteur* du bloc (*height*), soit par le résultat de *hachage* (ou *hash*) du bloc. Tout d'abord, trouvons un bloc par sa hauteur. Dans [???](#cup_of_coffee), nous avons vu que la transaction d'Alice était incluse dans le bloc 277316.

Nous utilisons la commande getblockhash, qui prend la hauteur du bloc comme paramètre et renvoie le hachage pour ce bloc :

$ bitcoin-cli getblockhash 277316 0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

Maintenant que nous savons dans quel bloc la transaction d'Alice était incluse, nous pouvons interroger ce bloc. Nous utilisons la commande getblock avec le hachage de bloc comme paramètre :

$ bitcoin-cli getblock 0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b3↵ 1b2cc7bdc4

{ "hash": "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4", "confirmations": 37371, "size": 218629, "height": 277316, "version": 2, "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e", "tx": [ "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f", "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe", "04905ff987ddd4cfe603b03cfb7ca50ee81d89d1f8f5f265c38f763eea4a21fd", "32467aab5d04f51940075055c2f20bbd1195727c961431bf0aff8443f9710f81", "561c5216944e21fa29dd12aaa1a45e3397f9c0d888359cb05e1f79fe73da37bd", [... des centaines de transactions ...] "78b300b2a1d2d9449b58db7bc71c3884d6e0579617e0da4991b9734cef7ab23a", "6c87130ec283ab4c2c493b190c20de4b28ff3caf72d16ffa1ce3e96f2069aca9", "6f423dbc3636ef193fd8898dfdf7621dcade1bbe509e963ffbff91f696d81a62", "802ba8b2adabc5796a9471f25b02ae6aeee2439c679a5c33c4bbcee97e081196", "eaaf6a048588d9ad4d1c092539bd571dd8af30635c152a3b0e8b611e67d1a1af", "e67abc6bd5e2cac169821afc51b207127f42b92a841e976f9b752157879ba8bd", "d38985a6a1bfd35037cb7776b2dc86797abbb7a06630f5d03df2785d50d5a2ac", "45ea0a3f6016d2bb90ab92c34a7aac9767671a8a84b9bcce6c019e60197c134b", "c098445d748ced5f178ef2ff96f2758cbec9eb32cb0fc65db313bcac1d3bc98f" ], "time": 1388185914, "mediantime": 1388183675, "nonce": 924591752, "bits": "1903a30c", "difficulty": 1180923195.258026, "chainwork": "000000000000000000000000000000000000000000000934695e92aaf53afa1a", "previousblockhash": "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569", "nextblockhash": "000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7" }

Le bloc contient 419 transactions et la 64ème transaction répertoriée (0627052b...) est le paiement du café d'Alice. L'entrée + hauteur + nous indique qu'il s'agit du 277316e bloc de la blockchain.

### Utilisation de l'interface de programmation de Bitcoin Core

L'assistant bitcoin-cli est très utile pour explorer la bibliothèue (API) Bitcoin Core et les fonctions de test. Mais tout l'intérêt d'une interface de programmation d'application est d'accéder aux fonctions par programmation. Dans cette section, nous montrerons comment accéder à Bitcoin Core à partir d'un autre programme.

L'API de Bitcoin Core est une interface JSON-RPC. JSON signifie JavaScript Object Notation et c'est un moyen très pratique de présenter des données que les humains et les programmes peuvent facilement lire. RPC signifie Remote Procedure Call (ou appel de procédure à distance), ce qui signifie que nous appelons des procédures (fonctions) distantes (sur le nœud Bitcoin Core) via un protocole réseau. Dans ce cas, le protocole réseau est HTTP ou HTTPS (pour les connexions cryptées).

Lorsque nous avons utilisé la commande bitcoin-cli pour obtenir de l'aide sur une commande, elle nous a montré un exemple d'utilisation de curl, le client HTTP polyvalent en ligne de commande pour construire l'un de ces appels JSON-RPC :

$ curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getblockchaininfo", "params": [] }' -H 'content-type: text/plain;' http://127.0.0.1:8332/

Cette commande montre que curl soumet une requête HTTP à l'hôte local (127.0.0.1), se connecte au port bitcoin par défaut (8332) et soumet une requête jsonrpc pour la méthode getblockchaininfo en utilisant l'encodage text/plain.

Vous remarquerez peut-être que curl demandera que les informations d'identification soient envoyées avec la demande. Bitcoin Core créera un mot de passe aléatoire à chaque démarrage et le placera dans le répertoire de données sous le nom .cookie. L'assistant bitcoin-cli peut lire ce fichier de mots de passe étant donné le répertoire de données. De même, vous pouvez copier le mot de passe et le transmettre à curl (ou à tout wrapper Bitcoin Core RPC de niveau supérieur). Alternativement, vous pouvez créer un mot de passe statique avec le script d'assistance fourni dans *./share/rpcauth/rpcauth.py* dans le répertoire source de Bitcoin Core.

Si vous implémentez un appel JSON-RPC dans votre propre programme, vous pouvez utiliser une bibliothèque HTTP générique pour construire l'appel, similaire à ce qui est montré dans l'exemple curl précédent.

Cependant, il existe des bibliothèques dans la plupart des langages de programmation qui "enveloppent" l'API Bitcoin Core d'une manière qui rend cela beaucoup plus simple. Nous utiliserons la bibliothèque python-bitcoinlib pour simplifier l'accès à l'API. N'oubliez pas que cela nécessite que vous disposiez d'une instance Bitcoin Core en cours d'exécution, qui sera utilisée pour effectuer des appels JSON-RPC.

Le script Python dans [example\_title](#rpc_example) effectue un simple appel getblockchaininfo et imprime le paramètre blocks à partir des données renvoyées par Bitcoin Core (nœud complet requis).

from bitcoin.rpc import RawProxy  
  
# Create a connection to local Bitcoin Core node  
p = RawProxy()  
  
# Run the getinfo command, store the resulting data in info  
info = p.getinfo()  
  
# Retrieve the 'blocks' element from the info  
print(info['blocks'])

Son exécution nous donne le résultat suivant:

$ python rpc\_example.py  
394075

Il nous indique que notre nœud local Bitcoin Core a 394075 blocs dans sa chaîne de blocs. Ce n'est pas un résultat spectaculaire, mais cela démontre l'utilisation de base de la bibliothèque en tant qu'interface simplifiée avec l'API JSON-RPC de Bitcoin Core.

Ensuite, utilisons les appels getrawtransaction et decodetransaction pour récupérer les détails du paiement du café d'Alice. Dans [example\_title](#rpc_transaction), nous récupérons la transaction d'Alice et listons les sorties de la transaction. Pour chaque sortie, nous affichons l'adresse et la valeur du destinataire. Pour rappel, la transaction d'Alice avait une sortie pour payer Bob's Cafe et une sortie pour rendre la monnaie à Alice.

from bitcoin.rpc import RawProxy  
  
p = RawProxy()  
  
# Alice's transaction ID  
txid = "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2"  
  
# First, retrieve the raw transaction in hex  
raw\_tx = p.getrawtransaction(txid)  
  
# Decode the transaction hex into a JSON object  
decoded\_tx = p.decoderawtransaction(raw\_tx)  
  
# Retrieve each of the outputs from the transaction  
for output in decoded\_tx['vout']:  
 print(output['scriptPubKey']['addresses'], output['value'])

En exécutant ce code, nous obtenons :

$ python rpc\_transaction.py  
([u'1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA'], Decimal('0.01500000'))  
([u'1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK'], Decimal('0.08450000'))

Les deux exemples précédents sont assez simples. Vous n'avez pas vraiment besoin d'un programme pour les exécuter ; vous pouvez tout aussi bien utiliser l'assistant bitcoin-cli. L'exemple suivant, cependant, nécessite plusieurs centaines d'appels RPC et démontre plus clairement l'utilisation d'une interface de programmation.

Dans [example\_title](#rpc_block), nous récupérons d'abord le bloc 277316, puis récupérons chacune des 419 transactions en référence à chaque identifiant de transaction. Ensuite, nous parcourons chacune des sorties de la transaction et additionnons la valeur.

from bitcoin.rpc import RawProxy  
  
p = RawProxy()  
  
# The block height where Alice's transaction was recorded  
blockheight = 277316  
  
# Get the block hash of block with height 277316  
blockhash = p.getblockhash(blockheight)  
  
# Retrieve the block by its hash  
block = p.getblock(blockhash)  
  
# Element tx contains the list of all transaction IDs in the block  
transactions = block['tx']  
  
block\_value = 0  
  
# Iterate through each transaction ID in the block  
for txid in transactions:  
 tx\_value = 0  
 # Retrieve the raw transaction by ID  
 raw\_tx = p.getrawtransaction(txid)  
 # Decode the transaction  
 decoded\_tx = p.decoderawtransaction(raw\_tx)  
 # Iterate through each output in the transaction  
 for output in decoded\_tx['vout']:  
 # Add up the value of each output  
 tx\_value = tx\_value + output['value']  
   
 # Add the value of this transaction to the total   
 block\_value = block\_value + tx\_value  
  
print("Total value in block: ", block\_value)

En exécutant ce code, nous obtenons :

$ python rpc\_block.py  
  
('Total value in block: ', Decimal('10322.07722534'))

Notre exemple de code calcule que la valeur totale des transactions dans ce bloc est de 10 322,07722534 BTC (dont 25 BTC de récompense et 0,0909 BTC de frais). Comparez cela au montant rapporté par un site d'exploration de blocs en recherchant le hachage ou la hauteur du bloc. Certains explorateurs de blocs rapportent la valeur totale hors récompense et hors frais. Voyez si vous pouvez repérer la différence.

## Clients, bibliothèques et boîtes à outils alternatifs

Il existe de nombreux clients alternatifs, bibliothèques, boîtes à outils et même des implémentations de nœuds complets dans l'écosystème bitcoin. Ceux-ci sont implémentés dans une variété de langages de programmation, offrant aux programmeurs des interfaces natives dans leur langue préférée.

Les sections suivantes répertorient certaines des meilleures bibliothèques, clients et kits d'outils, organisés par langages de programmation.

### C/C++

[Bitcoin Core](https://github.com/bitcoin/bitcoin)

L'implémentation de référence de bitcoin

[libbitcoin](https://github.com/libbitcoin/libbitcoin-system)

Boîte à outils de développement C++ multiplateforme, nœud et bibliothèque de consensus

[bitcoin explorer](https://github.com/libbitcoin/libbitcoin-explorer)

L'outil de ligne de commande de Libbitcoin

[picocoin](https://github.com/jgarzik/picocoin)

Bibliothèque cliente légère en langage C pour bitcoin par Jeff Garzik

### JavaScript

[bcoin](https://bcoin.io/)

Une implémentation modulaire et évolutive de nœud complet avec API

[Bitcore](https://bitcore.io/)

Noeud complet, API et bibliothèque par Bitpay

[BitcoinJS](https://github.com/bitcoinjs/bitcoinjs-lib)

Une pure bibliothèque Bitcoin en JavaScript pour node.js et les navigateurs

### Java

[bitcoinj](https://bitcoinj.github.io)

Une bibliothèque client Java à nœud complet

====PHP [bitwasp/bitcoin](https://github.com/bit-wasp/bitcoin-php):: Une bibliothèque Bitcoin en PHP et des projets associés

====Python [python-bitcoinlib](https://github.com/petertodd/python-bitcoinlib):: Une bibliothèque Bitcoin en Python dont une bibliothèque de consensus et un nœud par Peter Todd [pycoin](https://github.com/richardkiss/pycoin):: Une bibliothèque Bitcoin en Python par Richard Kiss [pybitcointools](https://github.com/primal100/pybitcointools):: Une fourche archivé de la bibliothèque Bitcoin en Python par Vitalik Buterin

### Ruby

[bitcoin-client](https://github.com/sinisterchipmunk/bitcoin-client)

Un wrapper de bibliothèque Ruby pour l'API JSON-RPC

### Go

[btcd](https://github.com/btcsuite/btcd)

Un client Bitcoin à nœud complet en langage Go

### Rust

[rust-bitcoin](https://github.com/rust-bitcoin/rust-bitcoin)

Bibliothèque Bitcoin Rust pour la sérialisation, l'analyse et les appels d'API

### C#

[NBitcoin](https://github.com/MetacoSA/NBitcoin)

Bibliothèque Bitcoin complète pour le framework .NET

### Objective-C

[CoreBitcoin](https://github.com/oleganza/CoreBitcoin)

Boîte à outils Bitcoin pour ObjC et Swift

De nombreuses autres bibliothèques existent dans une variété d'autres langages de programmation et d'autres sont créées tout le temps.