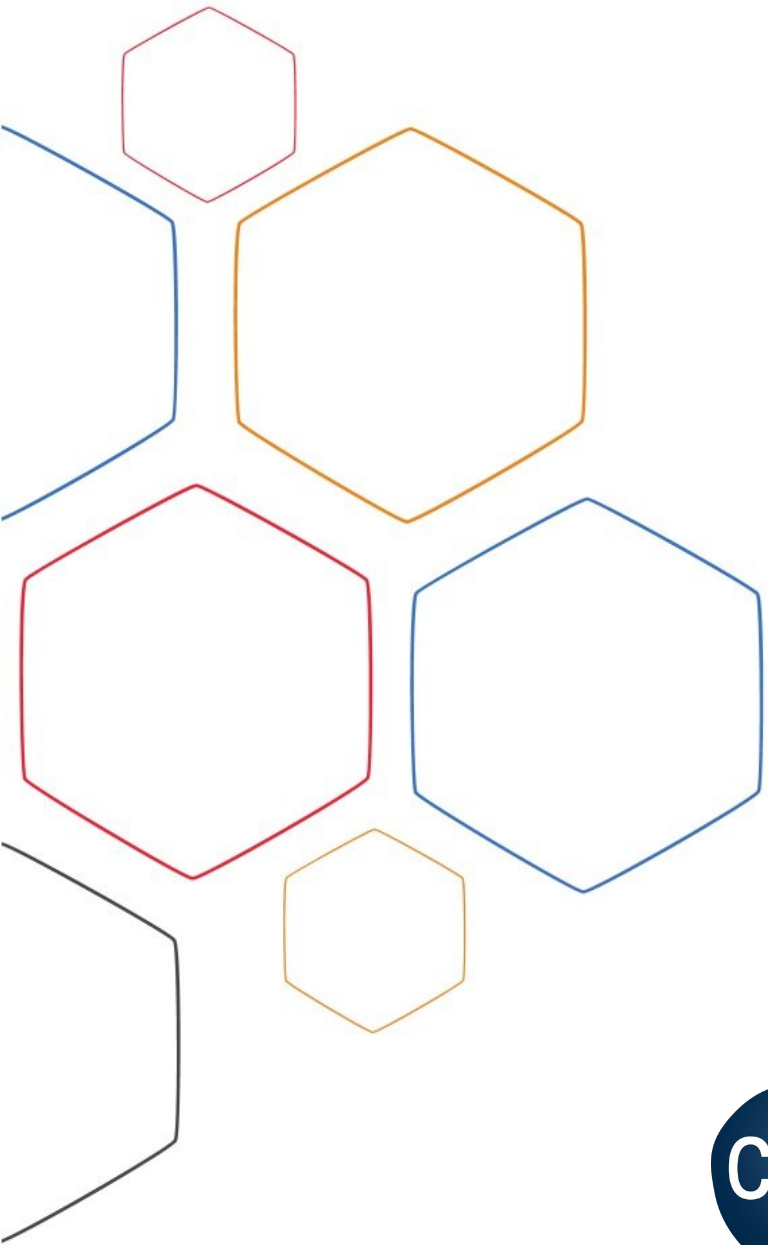
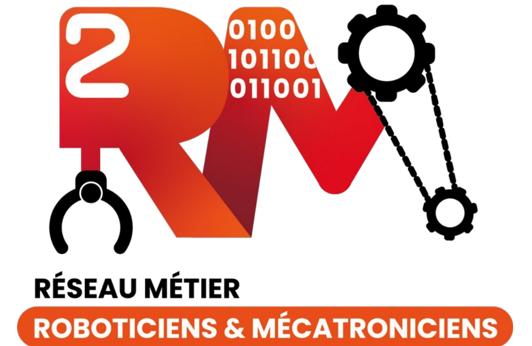




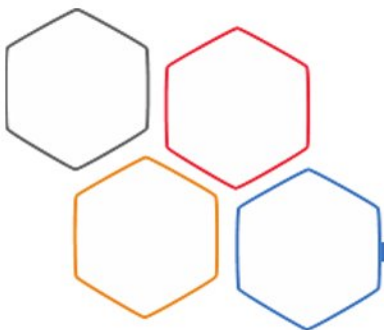
Introduction au packaging Conan

Robin Passama, Ingénieur de recherche CNRS, LIRMM

École Technologique 2RM, Rennes, 22-23 mai 2024

- cf. ma présentation de 2023
  - Pourquoi/Comment bien gérer le packaging
  - Pourquoi la communauté 2RM devrait s’y intéresser
- Trouver un système de packaging adapté aux besoin des roboticiens
  - Gestion des **langages compilés** (C, C++, CUDA, Rust…) qui sont la base de toutes nos applications
    - Packages binaires / compatibilité binaire
    - Cross-compilation
  - Idéalement une bonne gestion de langages de scripts (e.g. Python)
  - Gestion **simple** de contraintes de versions **complexes**
    - Utilisation de versions ***systèmes*** (plus généralement de version ***imposées***)

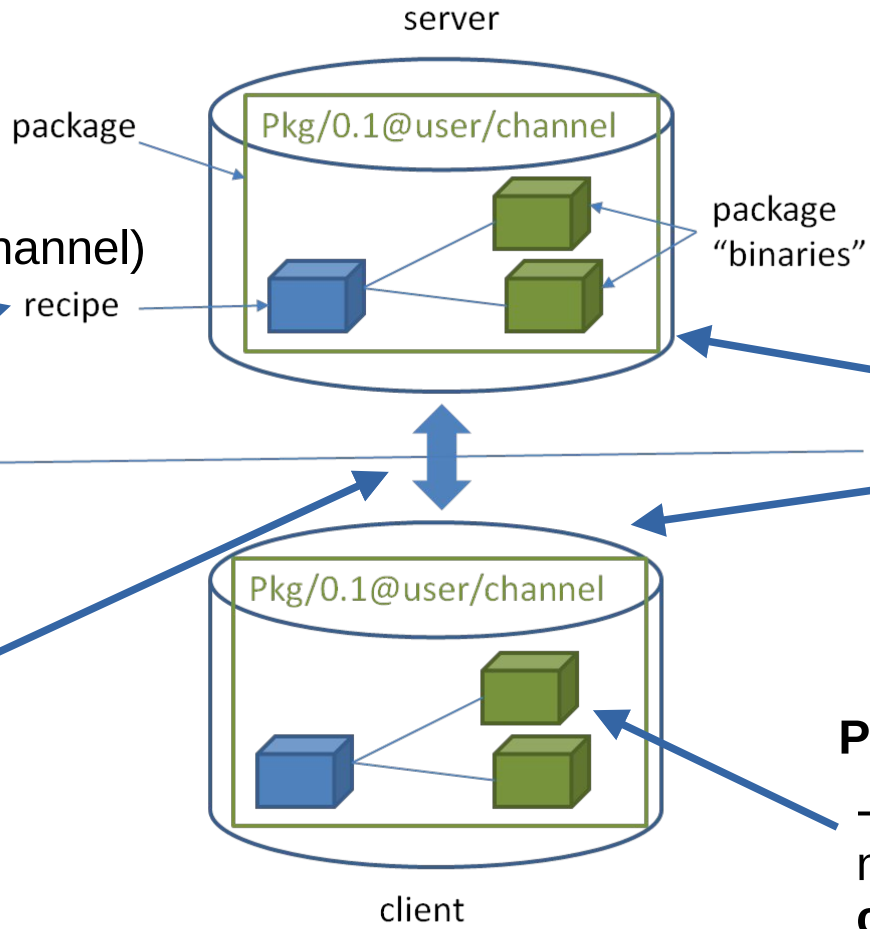
- Pourquoi lui ? De ce que je sais de prime abord :
  - Devenu « main-stream » en C/C++ (avec vcpkg)
    - Grosse communauté, équipe de développement dédiée
    - Beaucoup de projets déjà gérés
  - Libre
  - Très riche : flexible (en théorie adaptable à plusieurs langages compilés), très « scalable » ; « customisable » à souhait ; cross-(platform/build tools/IDE)
- Mais …
  - Il est réputé pour être difficile à prendre en main.
  - A priori pas de réel support Python
- Tant qu’on a pas essayé ...



Principes

Package
== identifiant unique

- **Projet** et **version(s)** cible(s)
- informations de variante (user,channel)



Recette (conanfile.py)

- Comment obtenir les sources
- Comment **builder** le projet
- Déclaration des **dépendances**

Package Registry

- distant : Conan Center, JFrogArtifactory, serveur Gitlab, etc.
- local : **cache** sur la station de travail

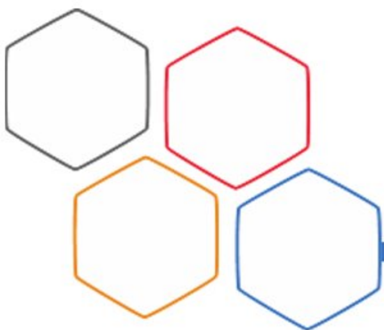
Relation client/server de type push/pull

- « à la git »

Package binaires enregistrés

- informations d’identification nécessaires à la **résolution de compatibilité binaire**

Credits: https://docs.conan.io/



Fonctionnement

- résoudre les dépendances d'un projet
- configurer le build

recette

profil

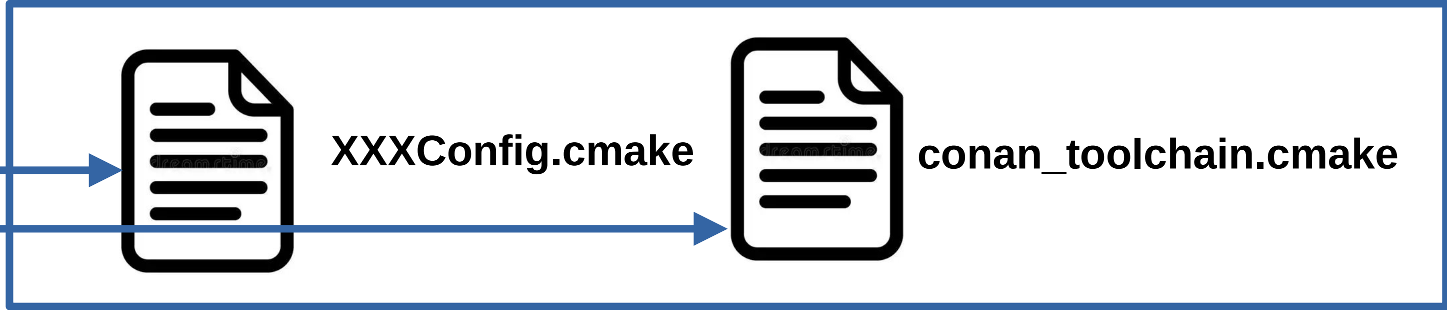
- toolchain (compilateur, librairie standard, etc.)
- description de la plateforme cible (os, architecture du processeur)
- type de build (Debug, Release, etc.)

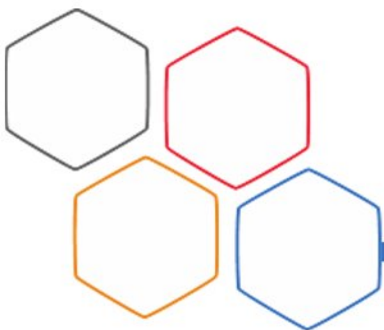
utilise

`conan install`

generators

- générer les fichiers nécessaires à des outils tiers (e.g. CMake) :
  - résolution des dépendances
  - configuration du build





Résolution des dépendances

server

A**2**) recette existe dans un remote ?

package

Pkg/0.1@user/channel

package "binaries"

OUI

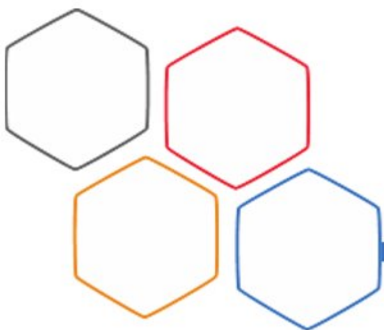
NON

A**3**) télécharge la recette

Pkg/0.1@user/channel

A**1**) existe pas dans le cache local ?

client



Résolution des dépendances

server

package

recipe

Pkg/0.1@user/channel

package "binaries"

B2) binaire compatible ?

OUI

B3) télécharge le binaire

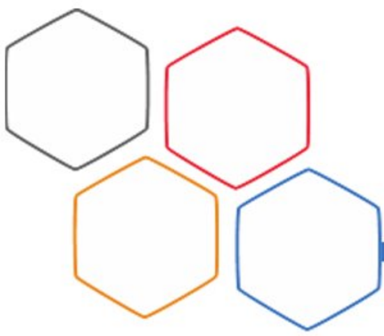
NON

Pkg/0.1@user/channel

B**4**) builde la recette

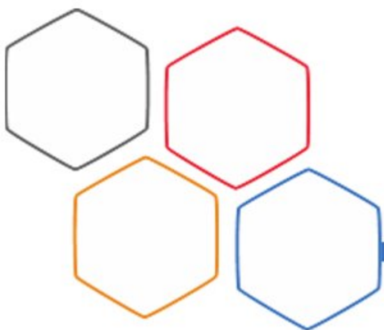
B**1**) binaire compatible ?

client



Configurer le build, pourquoi ?

- Utiliser une même configuration globale pour plusieurs projets
  - Fichier `profile` contient les informations communes
  - Transformé en configuration d’une *toolchain*
- Permet de « garantir » la compatibilité binaire
  - Profil + Recette appliqué au **build du package local**
  - Profil + Recette utilisés pour configurer chaque dépendance
    - Soit utiliser un **binaire compatible**
    - Soit **builder** le projet



Fonctionnement

- « enregistrer » un projet comme un package Conan
- contrôler/automatiser le build du projet

recette

profil

`conan create`

Le projet peut être utilisé comme **dépendance via conan**

XXXConfig.cmake

conan_toolchain.cmake

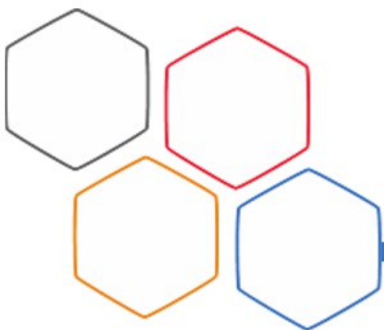
\+

Obtention des sources du projet:
- git
- téléchargement
- patch
- ...

\+

build

Installation dans le cache



Recette

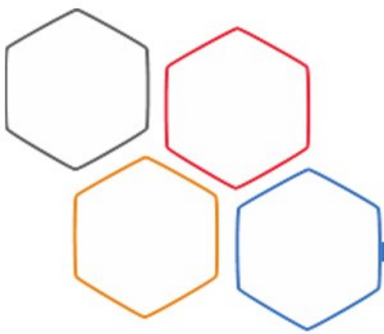
```
from conan import ConanFile
from conan.tools.cmake import CMakeToolchain, CMake, cmake_layout, CMakeDeps


class helloRecipe(ConanFile):
    name = "hello"
    version = "1.0"


    package_type = "library"
```

Déclaration du package

Type de contenu

documentation

```
    # Optional metadata
    license = "<Put the package license here>"
    author = "<Put your name here> <And your email here>"
    url = "<Package recipe repository url here, for issues about the package>"
    description = "<Description of hello package here>"
    topics = ("<Put some tag here>", "<here>", "<and here>")
```



Recette

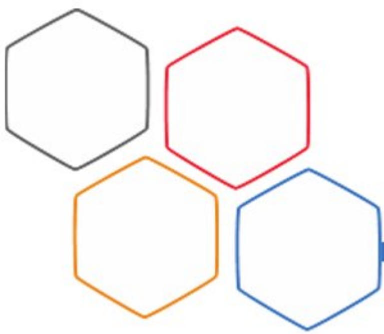
Informations pour la compatibilité binaire : binaire compatible si même OS, compilateur, build type et architecture processeur.

```
settings = "os", "compiler", "build_type", "arch"
options = {"shared": [True, False], "fPIC": [True, False]}
```

```
default_options = {"shared": False, "fPIC": True}
```

Generators utilisés : Le projet est un projet CMake

```
generators = "CMakeToolchain", "CMakeDeps"
```



Recette

Cas 1 : Package Conan « natif » (projet CMake)

```
exports_sources = "CMakeLists.txt", "src/*", "include/*"
```

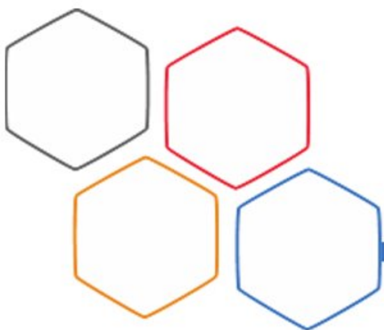
Cas 2 : Télécharger des sources existantes

```
from conan.tools.files import get
    ...
    def source(self):
        get(self, "https://github.com/conan-io/libhello/archive/refs/heads/main.zip",
                  strip_root=True)
```

Cas 3 : Cloner un dépôt existant

```
from conan.tools.scm import Git
    ...
    def source(self):
        git = Git(self)
        git.clone(url="https://github.com/conan-io/libhello.git", target=".")
```

… possibilité de patcher/remplacer des fichiers



Recette

Optionnel

```
def layout(self):
    cmake_layout(self)
```

→ Convention de structuration

```
def config_options(self):
    if self.settings.os == "Windows":
        del self.options.fPIC
```

→ Actions spécifiques sur les **options** du package

```
def generate(self):
    tc = CMakeToolchain(self)
    tc.generate()
```

→ Customisation de l'appel du **generator** de toolchain

```
def build(self):
    cmake = CMake(self)
    cmake.configure()
    cmake.build()
```

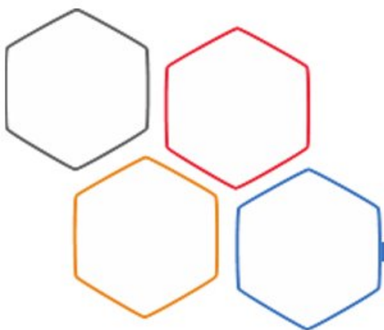
→ Automatisation de la construction du projet

```
def package(self):
    cmake = CMake(self)
    cmake.install()
```

→ Installation du projet (création du package binaire)

```
def package_info(self):
    self.cpp_info.libs = ["hello"]
```

→ Information exportée aux utilisateurs (e.g. targets, flags)



Recette : dépendances

Optionnel

```python
def requirements(self):
    self.requires("zlib/1.2.11")
    self.requires("openssl/1.1.1k")
```

Dépendances **fonctionnelles** : run-time, compile-time et link-time

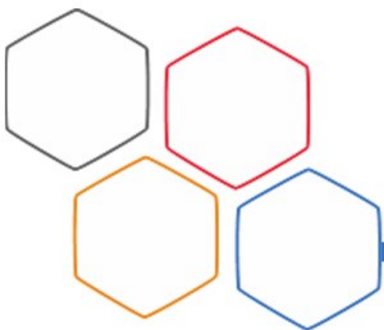
```python
def build_requirements(self):
    self.tool_requires("cmake/3.18.4")
    self.tool_requires("ninja/1.10.2")
```

Dépendances d’**outils** : utilisées uniquement pendant le build (pas exportées)

```python
def test_requirements(self):
    self.test_requires("catch2/2.13.4")
```

Dépendances de **test** : utilisées uniquement pendant le test (pas exportées)

Tout est *packagable* en Conan, y compris les **toolchains** !



Recette : versionnement

- Contraintes de version
  - Stricte :

```
self.requires("zlib/1.2.11")
```

  - Intervalles :

```
self.requires("zlib/[>=1.2.8 <1.3]")
```

  - Approximées
    - Dernier nombre fixe, e.g. n’importe quelle 1.2.X

```
self.requires("zlib/[~1.2]") # 1.2.0, 1.2.3 OK, 1.3 KO
```

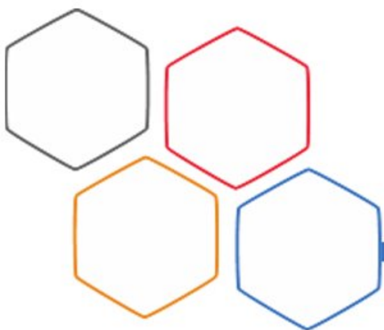
    - Dernier nombre variable :

```
self.requires("zlib/[^1.2]") #1.2.3, 1.8.6, 1.8.6 OK, 1.0, 2.0 KO
```

  - Union de tout ça :

```
self.requires("zlib/[^1.2] || 2.0")
```

Override, force : https://docs.conan.io/2/tutorial/versioning/conflicts.html



Profil

`conan install . --build=missing --profile=`**`someprofile`**

```
[settings]
arch=x86_64
build_type=Release
compiler=gcc
compiler.cppstd=gnu17
compiler.libcxx=libstdc++11
compiler.version=11
os=Linux

[options]
*:shared=True

[tool_requires]
*:cmake/2.24@robin/lirmm

[buildenv]
CC=gcc11
. . .
```

Spécificités de ma station de travail, de mon projet

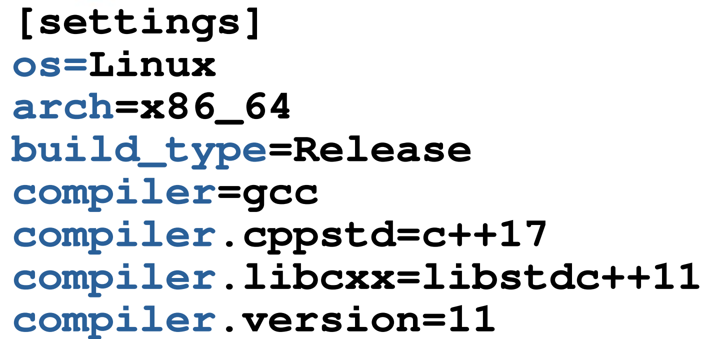
Options de projets par défaut

Version de CMake utilisée par défaut

Variables d’environnement utilisées pendant le build

Profil par défaut peut être détecté/généré

Profil : contextes et cross compilation



default

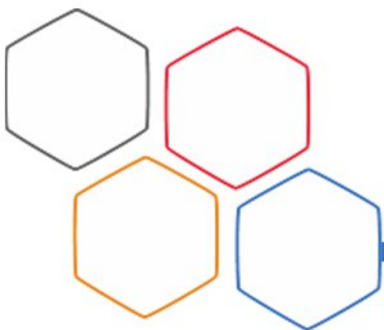
```
[settings]
os=Linux
arch=armv7hf
build_type=Release
compiler=gcc
compiler.cppstd=gnu14
compiler.libcxx=libstdc++11
compiler.version=9
[buildenv]
CC=arm-linux-gnueabihf-gcc-9
CXX=arm-linux-gnueabihf-g++-9
LD=arm-linux-gnueabihf-ld
```

raspberry

Pour gérer les dépendances de **build**

Pour gérer les dépendances

```
conan install . --build=missing --profile:build=default --profile:host=raspberry
```



Gestion des remotes

- Remote = Repository distant
  - Remote par défaut : **Conan Center**
  - Plusieurs remotes possibles
  - Upload « à la main » :
    - Dernière révision (dernier build)

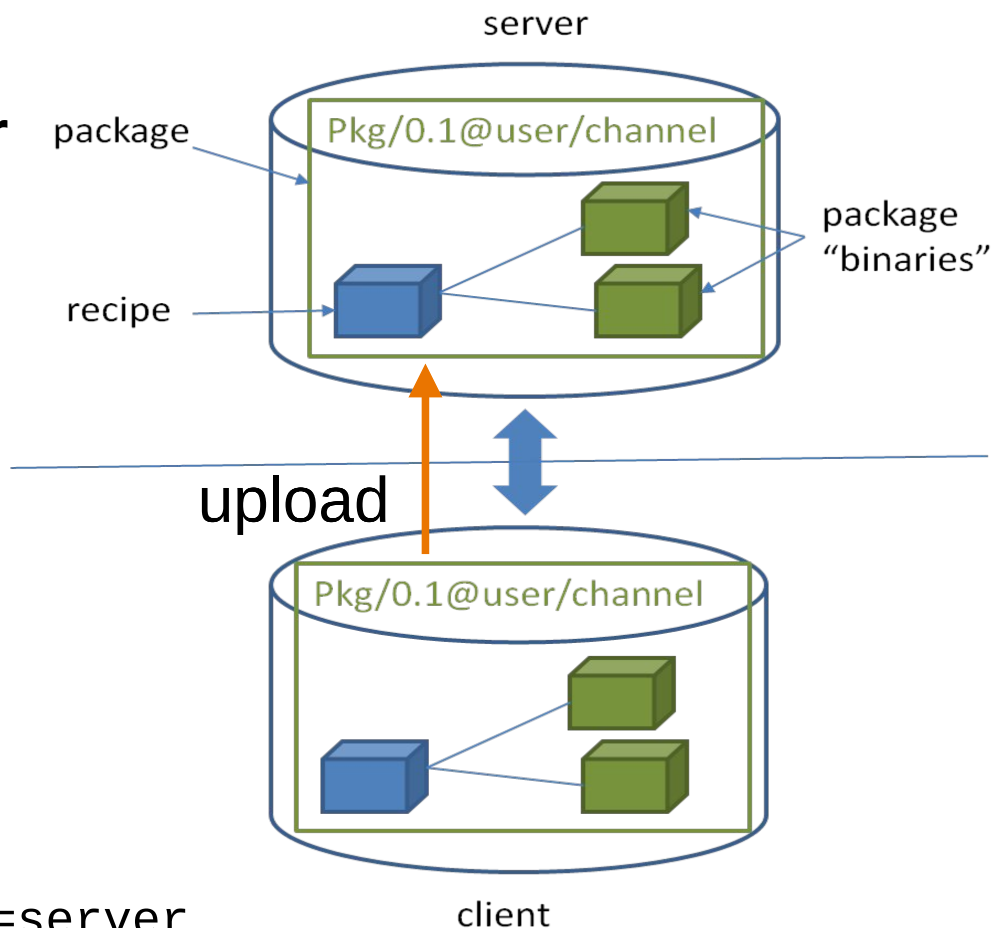
```
conan upload Pkg/0.1 -r=server
```

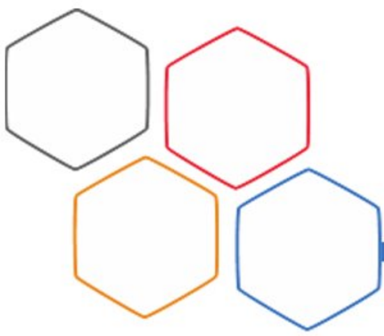
    - Tous les binaires

```
conan upload Pkg/0.1:* -r=server
```

    - Uniquement la recette

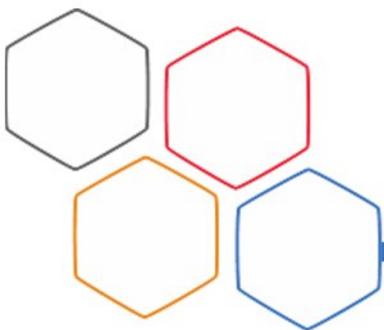
```
conan upload Pkg/0.1 --only-recipe -r=server
```





Pourquoi tout ça ?

- Garantie globale de **compatibilité binaire** !!
- Tout se fait **automatiquement** « à la volée »
  - Installation/recherche des dépendances
  - Résolution des versions (si faisable)
  - Téléchargement de binaires/utilisation du cache local
- Facilité de **changement global de configuration**
  - plateforme cible, mode de build, etc.
- Possibilités de **customisation** à tous les niveaux, par exemple
  - Au lieu de builder, déployer un package binaire (e.g. apt)
  - Supporter de nouveaux build system



Pourquoi tout ça ?

- Rendre les build system **compatibles** sans effort

autoconf
User1

pkconfig

CMake
User2

find_package

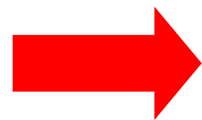
CMake
MyProject

find_package

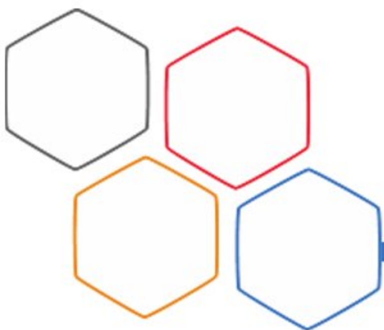
autoconf
Dep1

find_package

CMake
Dep2

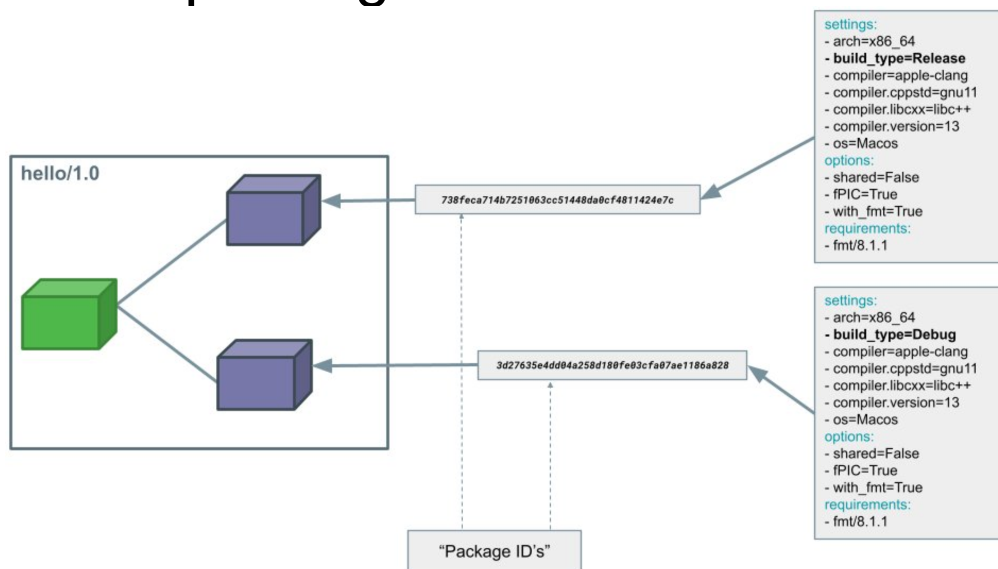


Génération automatisée « à la demande » des fichiers de configuration des dépendances

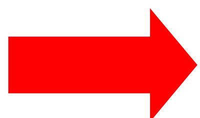


Comment ça marche

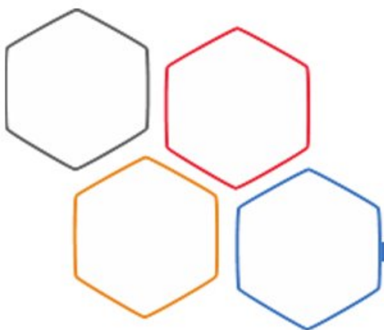
- Le package ID
  - Hash (SHA1) unique représentant un package binaire
  - Construit à partir de :
    - **Settings**
    - **Options**
    - *Dépendances*



Credits :https://docs.conan.io/



Sert pour les recherches de packages binaire compatibles

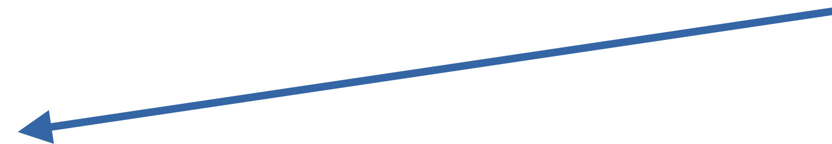


Comment ça marche

- Les révisions

```
conan list hello/1.0:*
```

```
Local Cache
 hello
    hello/1.0
      revisions
        54bbf3d9749e5e4839d5554264d99d64 (2024-04-29 16:44:05 UTC)
          packages
            4b536c37735510182960b2b41fe7dc83c79147b5
              info
                settings
                  arch: x86_64
                  build_type: Release
                  compiler: gcc
                  compiler.cppstd: gnu17
                  compiler.libcxx: libstdc++11
                  compiler.version: 11
                  os: Linux
                options
                  fPIC: True
                  shared: False
                  with_fmt: True
                requires
                  fmt/8.1.Z
```



Révision : modifications de la recette / des sources

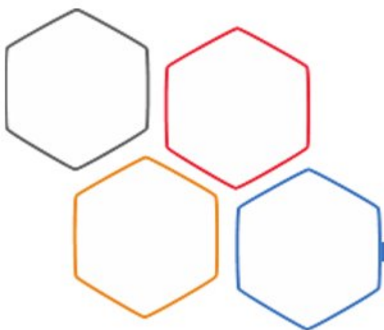
Package ID

Fonctionnalitées non abordées

- Les variantes de recettes
  - Variante par utilisateur et « branche »
- Builds reproductibles
  - Utiliser les **révisions** de package et les fichiers **lockfile**
- Les packages hybrides, type compilateurs ou générateurs de code (e.g. protobuf)
  - Définissent : exécutable(s) + librairie(s)
  - Utilisés comme : build dependency et normal dependency
- Définir des toolchains comme des packages conan
- Etc.

Les sujets qui fâchent

- Support **Python** théoriquement possible (sans trop se casser la tête) mais :
  - Pas de package disponible par défaut dans le conan center
  - Conan est **overkill** pour packager du Python, **sauf pour les bindings** !!
- Support de **langages compilés** (CUDA, Fortran) théoriquement possible mais :
  - Langages autres que C++ pas supportés nativement
    - Toujours possible d’écrire des solutions non natives (moins portables)
  - Pas de package disponible par défaut dans le conan center
- Pas de support **ROS2**
  - Tentative de packager ROS2 avec Conan a apparemment été abandonnée
- Gitlab ne gère pas les **repository** conan 2 !
  - Ne semble pas être une priorité



Merci pour votre attention