Parfait. Voici le contenu **Markdown** prêt à copier dans un fichier `.md` (ex: `macos_vm_fix.md`).

```
# macOS Sonoma (OSX-KVM / QEMU / WSL2) - Problèmes fréquents & Solutions

## Contexte
- Host : Windows + WSL2 (Ubuntu)
- VM : macOS Sonoma via OSX-KVM (QEMU + KVM)
- Objectif : installer outils de build (clang/Xcode) + utiliser macOS pour
développer iOS/macOS/tvOS/watchOS

---

# 1) Problème : Impossible de se connecter aux serveurs Apple

## Symptômes
- `xcode-select --install` échoue :
  > Can’t connect to the Apple Software Update server.
- Homebrew échoue lors de l’installation des CLT
- `curl -I https://developer.apple.com` renvoie :
  > curl: (52) Empty reply from server
- `sntp time.apple.com` :
  > Timeout

Alors que :
- `ping 8.8.8.8` fonctionne
- `ping google.com` fonctionne

## Cause probable
Le réseau de la VM tente d'utiliser **IPv6**, mais IPv6 est mal supporté /
instable dans le setup WSL2 + QEMU NAT.

---

# 2) Diagnostic

## Test IPv4
Tester la connexion Apple en forçant IPv4 :

```bash
curl -4I https://developer.apple.com
```

Résultat attendu

Si on obtient un `HTTP/1.1 200 OK`, alors IPv4 fonctionne.
=> Le problème vient de IPv6.

---

3) Solution : Désactiver IPv6 sur macOS (Ethernet)
```

Étape 1 : voir les services réseau

```
networksetup -listallnetworkservices
```

Exemple sortie :

- Ethernet
- Wi-Fi
- Thunderbolt Bridge

Étape 2 : désactiver IPv6 sur Ethernet

```
sudo networksetup -setv6off Ethernet
```

(Si le service ne s'appelle pas Ethernet, remplacer par le nom exact.)

Étape 3 : vérifier la config

```
networksetup -getinfo Ethernet
```

---

4) Bonus : Désactiver IPv6 côté QEMU

Dans le script `OpenCore-Boot.sh`, modifier :

Avant :

```
-netdev user,id=net0 \
```

Après :

```
-netdev user,id=net0,ipv6=off \
```

Cela empêche macOS de recevoir IPv6.

---

5) Après correction : Installer les outils de compilation (clang)

Installer Command Line Tools (recommandé)

```
xcode-select --install
```

Cela installe :

- clang
- clang++
- lldb
- make
- git
- headers / SDK macOS

Vérifier :

```
clang --version
xcode-select -p
```

6) Installer Homebrew (optionnel mais utile)

Installer :

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Après installation :

```
brew --version
```

7) Installer Xcode complet (nécessaire pour iOS/tvOS/watchOS)

⚠ Les Command Line Tools suffisent pour compiler des apps macOS CLI,
mais pour iOS/watchOS/tvOS il faut le SDK correspondant, donc Xcode complet.

Installation via App Store

1. Ouvrir App Store
2. Rechercher "Xcode"
3. Installer

Vérifier

```
xcodebuild -version
xcodebuild -showsdks
```

---

8) Activer Xcode après installation

Après installation de Xcode :

```
sudo xcode-select -s /Applications/Xcode.app/Contents/Developer
sudo xcodebuild -license accept
```

---

9) Compilation Objective-C++ (.mm) pour macOS (Terminal)

Exemple simple :

```
clang++ -std=c++20 main.mm -framework Cocoa -o app
```

Exécuter :

```
./app
```

---

10) Commande personnalisée pour démarrer macOS VM depuis WSL

Créer un script dans WSL :

```
nano ~/rihenmacos
```

Contenu :

```
#!/usr/bin/env bash
cd ~/OSX-KVM
./OpenCore-Boot.sh
```

Rendre exécutable :

```
chmod +x ~/rihenmacos
```

Installer globalement :

```
sudo cp ~/rihenmacos /usr/local/bin/rihenmacos
```

Utilisation :

```
rihenmacos
```

Notes

- macOS en VM n'a pas de vrai "Wi-Fi". On utilise Ethernet virtuel.
- Certains freezes peuvent venir du GPU virtuel.
- `-vga vmware` est souvent plus stable que virtio-vga pour macOS installer.

```
---

# Installer les outils clang pour macOS + iOS + tvOS + watchOS

### ☑ 1) Pour macOS uniquement
Tu peux utiliser juste :

- **Command Line Tools** (`xcode-select --install`)
```

```
Ça donne clang + SDK macOS.

---

### ☑ 2) Pour iOS/tvOS/watchOS (obligatoire)
Tu dois installer **Xcode complet**.

Parce que les SDK Apple sont inclus dedans :
- iPhoneOS.sdk
- iPhoneSimulator.sdk
- AppleTVOS.sdk
- WatchOS.sdk
- etc.

Après installation, vérifie que tu les as :

```bash
xcodebuild -showsdks
```

Tu verras par exemple :

- iphoneos
- iphonesimulator
- appletvos
- appletvsimulator
- watchos
- watchsimulator
- macosx

Compiler pour iOS depuis le terminal

Exemple (build iOS avec xcodebuild) :

```
xcodebuild -scheme MonApp -sdk iphoneos -configuration Release build
```

Pour simulateur :

```
xcodebuild -scheme MonApp -sdk iphonesimulator -configuration Debug build
```

Pour ton build system custom

Une fois Xcode installé, tu peux directement utiliser :

- `clang`
```

- `clang++`
- `xcrun`
- `xcodebuild`

Le plus propre est de passer par `xcrun` :

Exemple :

```
xcrun --sdk macosx clang++ main.mm -framework Cocoa -o app
```

Pour iOS :

```
xcrun --sdk iphoneos clang++ main.mm -o app
```

Mais attention : pour iOS, il faut un projet iOS complet (UIKit, entitlements, signing). Donc souvent tu compiles via `xcodebuild`.

Si tu veux, je peux te donner une structure complète de commandes pour compiler :

- macOS app Cocoa
- iOS app minimal
- library static/dynamic multi-platform avec `xcrun` + `lipo` + `xcodebuild` (style framework universel).

Voici une **structure complète** (dossiers + commandes) pour compiler depuis le terminal **staticlib / sharedlib / consoleapp / windowedapp** sur **macOS**, et pour produire des **libs multi-plateformes (iOS/tvOS/watchOS + simulateurs)** sous forme de **XCFramework**.

> Pour iOS/tvOS/watchOS : compiler une *app* "pure terminal" est possible mais vite pénible (bundle, Info.plist, signature, entitlements). En pratique, on build les apps via `xcodebuild`. Par contre, les **libs** (static/dynamic + XCFramework) se font très bien en CLI.

---

0) Arborescence recommandée

```
project/
  include/
    mylib.h
  src/
    mylib.cpp
    mylib.mm          (si besoin Cocoa / ObjC++)
    main.cpp          (console)
    app.mm            (windowed macOS)
  build/
    macos/
    ios/
```

```
    tvos/
    watchos/
  dist/
    lib/
    bin/
```

---

1) Variables communes (à utiliser dans tes scripts)

Récupérer les SDK paths

```
MACOS_SDK="$(xcrun --sdk macosx --show-sdk-path)"
IOS_SDK="$(xcrun --sdk iphoneos --show-sdk-path)"
IOSSIM_SDK="$(xcrun --sdk iphonesimulator --show-sdk-path)"
TVOS_SDK="$(xcrun --sdk appletvos --show-sdk-path)"
TVSIM_SDK="$(xcrun --sdk appletvsimulator --show-sdk-path)"
WATCHOS_SDK="$(xcrun --sdk watchos --show-sdk-path)"
WATCHSIM_SDK="$(xcrun --sdk watchsimulator --show-sdk-path)"
```

Choisir minimum OS (exemples)

```
MACOS_MIN=11.0
IOS_MIN=13.0
TVOS_MIN=13.0
WATCHOS_MIN=7.0
```

---

2) macOS

A) Console app (C++)

```
mkdir -p build/macos dist/bin

xcrun --sdk macosx clang++ -std=c++20 -O2 \
  -mmacosx-version-min=$MACOS_MIN \
  -Iinclude src/main.cpp \
  -o dist/bin/my_console_app
```

B) Windowed app (Cocoa) avec `.mm`

(Programme minimal Cocoa)

- compile/link avec framework Cocoa :

```
mkdir -p build/macos dist/bin

xcrun --sdk macosx clang++ -std=c++20 -O2 \
  -mmacosx-version-min=$MACOS_MIN \
  -Iinclude src/app.mm \
  -framework Cocoa \
  -o dist/bin/my_window_app
```

> Si tu veux une “vraie app bundle” `.app`, dis-le et je te donne la structure `MyApp.app/Contents/...` + Info.plist + codesign (c’est faisable, juste un peu plus long).

C) Static library `.a` (macOS)

```
mkdir -p build/macos dist/lib

# 1) compile objets
xcrun --sdk macosx clang++ -std=c++20 -O2 -fPIC \
  -mmacosx-version-min=$MACOS_MIN \
  -Iinclude -c src/mylib.cpp -o build/macos/mylib.o

# 2) archive
xcrun --sdk macosx ar rcs dist/lib/libmylib_macos.a build/macos/mylib.o

# (optionnel) index
xcrun --sdk macosx ranlib dist/lib/libmylib_macos.a
```

D) Shared library `.dylib` (macOS)

```
mkdir -p build/macos dist/lib

xcrun --sdk macosx clang++ -std=c++20 -O2 -fPIC \
  -mmacosx-version-min=$MACOS_MIN \
  -Iinclude -c src/mylib.cpp -o build/macos/mylib.o

xcrun --sdk macosx clang++ -shared \
  -mmacosx-version-min=$MACOS_MIN \
  -Wl,-install_name,@rpath/libmylib.dylib \
  -o dist/lib/libmylib.dylib build/macos/mylib.o
```

---

3) iOS / tvOS / watchOS : build de LIBS (recommandé) + XCFramework

A) Static lib iOS device (arm64)

```
```

```
mkdir -p build/ios/device dist/lib

xcrun --sdk iphoneos clang++ -std=c++20 -O2 -fPIC \
  -isysroot "$IOS_SDK" \
  -miphoneos-version-min=$IOS_MIN \
  -arch arm64 \
  -Iinclude -c src/mylib.cpp -o build/ios/device/mylib.o

xcrun --sdk iphoneos ar rcs dist/lib/libmylib_ios_arm64.a
build/ios/device/mylib.o
xcrun --sdk iphoneos ranlib dist/lib/libmylib_ios_arm64.a
```

B) Static lib iOS simulator (arm64 + x86_64) puis “fat” (un seul `.a`)

Sur Sonoma (Mac Apple Silicon), souvent arm64 suffit, mais je te donne les 2 archs :

```
mkdir -p build/ios/sim_arm64 build/ios/sim_x64 dist/lib

# sim arm64
xcrun --sdk iphonesimulator clang++ -std=c++20 -O2 -fPIC \
  -isysroot "$IOSSIM_SDK" \
  -mios-simulator-version-min=$IOS_MIN \
  -arch arm64 \
  -Iinclude -c src/mylib.cpp -o build/ios/sim_arm64/mylib.o

xcrun --sdk iphonesimulator ar rcs build/ios/libmylib_sim_arm64.a
build/ios/sim_arm64/mylib.o
xcrun --sdk iphonesimulator ranlib build/ios/libmylib_sim_arm64.a

# sim x86_64
xcrun --sdk iphonesimulator clang++ -std=c++20 -O2 -fPIC \
  -isysroot "$IOSSIM_SDK" \
  -mios-simulator-version-min=$IOS_MIN \
  -arch x86_64 \
  -Iinclude -c src/mylib.cpp -o build/ios/sim_x64/mylib.o

xcrun --sdk iphonesimulator ar rcs build/ios/libmylib_sim_x86_64.a
build/ios/sim_x64/mylib.o
xcrun --sdk iphonesimulator ranlib build/ios/libmylib_sim_x86_64.a

# fusion en fat lib (sim)
xcrun lipo -create \
  build/ios/libmylib_sim_arm64.a \
  build/ios/libmylib_sim_x86_64.a \
  -output dist/lib/libmylib_iossim.a
```

C) TVOS / watchOS

Même logique, juste flags version-min + sdk :

- tvOS device: `--sdk appletvos` + `-mappletvos-version-min=$TVOS_MIN`
- tvOS sim: `--sdk appletvsimulator` + `-mtvos-simulator-version-min=$TVOS_MIN`
- watchOS device: `--sdk watchos` + `-mwatchos-version-min=$WATCHOS_MIN`
- watchOS sim: `--sdk watchsimulator` + `-mwatchos-simulator-version-min=$WATCHOS_MIN`

(Je peux te donner les 4 blocs complets comme iOS, mais c'est exactement le même "template".)

---

4) Créer une XCFramework (le format moderne multi-plateformes)

Une fois que tu as tes libs et headers, fais :

```
mkdir -p dist/xcframework

xcodebuild -create-xcframework \
  -library dist/lib/libmylib_ios_arm64.a -headers include \
  -library dist/lib/libmylib_iossim.a -headers include \
  -output dist/xcframework/MyLib.xcframework
```

Tu peux ajouter tvOS/watchOS de la même façon :

```
xcodebuild -create-xcframework \
  -library dist/lib/libmylib_ios_arm64.a -headers include \
  -library dist/lib/libmylib_iossim.a -headers include \
  -library dist/lib/libmylib_tvos_arm64.a -headers include \
  -library dist/lib/libmylib_tvossim.a -headers include \
  -library dist/lib/libmylib_watchos_arm64.a -headers include \
  -library dist/lib/libmylib_watchsim.a -headers include \
  -output dist/xcframework/MyLib.xcframework
```

---

5) Apps iOS/tvOS/watchOS : build CLI (avec Xcode project)

Quand tu as un projet Xcode (ou un `.xcodeproj` généré), tu build comme ça :

iOS device

```
xcodebuild \
  -project MyApp.xcodeproj \
  -scheme MyApp \
  -sdk iphoneos \
  -configuration Release \
  -destination 'generic/platform=iOS' \
  build
```

iOS simulator

```
xcodebuild \
  -project MyApp.xcodeproj \
  -scheme MyApp \
  -sdk iphonesimulator \
  -configuration Debug \
  -destination 'platform=iOS Simulator,name=iPhone 15' \
  build
```

> La signature (codesign) peut bloquer sur device si tu n'as pas un compte configuré. Pour tester "sans device", reste sur simulateur.

---

6) Ta demande spécifique : C++ + `.mm` qui ouvre une fenêtre macOS

Oui : `.mm` te permet d'utiliser Cocoa (Objective-C) et appeler ton code C++.

Si tu veux, envoie-moi ton objectif (ex: "ouvrir une fenêtre + dessiner un triangle / intégrer Metal / juste un bouton"), et je te donne un `app.mm` minimal complet + build command.

Si tu me confirmes ton CPU (Apple Silicon ou Intel) et si tu veux supporter **x86_64 simulateur** ou seulement **arm64**, je te donne le script final "tout-en-un" (iOS + tvOS + watchOS + macOS) sans variantes inutiles.

Oui, ta stratégie est **bonne** : garder **90–99%** du code en `.h/.cpp` (moteur, rendu, logique, etc.) et n'avoir que des "points d'entrée / window / events / platform glue" en **fichiers spécifiques plateforme** (`.mm` sur macOS/iOS, `.cpp` Win32, X11/Wayland, Android NDK…). C'est exactement le pattern utilisé par beaucoup de moteurs.

Le seul truc à respecter : **ne mélange pas Cocoa/UIKit dans tes headers C++ publics**. Fais une couche "Platform" mince qui appelle ton core C++.

Architecture conseillée (propre et scalable)

1) Séparer "Core" et "Platform"

- `Core/` : pur C++ portable (no Apple headers)
- `Platform/<OS>/` : OS glue

Exemple :

```
src/Unkeny2D/
  Core/...
  Rendering/...
```

```
  Window/
    Window.h            (interface C++ portable)
    Window.cpp          (factory + logique commune)
  Platform/
    macOS/
      MacApp.mm         (NSApplicationMain / delegate)
      MacWindow.mm      (NSWindow + events -> C++)
    iOS/
      IOSApp.mm         (UIApplicationMain)
      IOSView.mm        (UIView / CAMetalLayer -> C++)
    Windows/
      WinMain.cpp
      WinWindow.cpp
    Linux/
      XcbWindow.cpp
```

2) Interface window/events en C++ pur

`Window.h` contient une interface neutre :

```cpp
class Window {
public:
  virtual void PollEvents() = 0;
  virtual void SetTitle(const char*) = 0;
  virtual ~Window() = default;
};
std::unique_ptr<Window> CreateWindow(const WindowDesc&);
```

Les `.mm` font uniquement :

- créer fenêtre Cocoa/UIKit
- convertir events → appeler callbacks C++ (KeyDown, MouseMove, Resize)

Ce que tu peux compiler en “CLI pur” vs “Xcodebuild”

☑ macOS

- **ConsoleApp** : CLI (clang++)
- **WindowedApp** Cocoa : CLI possible (`-framework Cocoa`) + tu peux même créer un `.app` bundle toi-même

☑ iOS / tvOS / watchOS

- **Libs** : CLI (xcrun + ar + lipo + xcframework) → parfait pour ton build system
- **Apps** : techniquement possible en CLI, mais **en pratique** tu passes par `xcodebuild` (bundle, assets, entitlements, signing, storyboard, etc.)

- Pour dev rapide : build **Simulator** sans te battre avec la signature.

Donc : ton build system peut très bien faire :

- `StaticLib/SharedLib` pour iOS/tvOS/watchOS en CLI
- et `WindowedApp` iOS/tvOS/watchOS via `xcodebuild` (en générant un .xcodeproj minimal ou en gardant un template projet).

Mapping direct vers tes enums (ProjectKind + TargetOS)

ProjectKind.CONSOLE_APP

- macOS/Linux/Windows : `clang++ main.cpp -o app`
- iOS/tvOS/watchOS : pas utile (pas de console classique)

ProjectKind.WINDOWED_APP

- macOS : `.mm` + Cocoa
- iOS/tvOS : `.mm` + UIKit
- watchOS : app watch = plus spécifique, souvent via template Xcode (je conseille `xcodebuild`)

ProjectKind.STATIC_LIB / SHARED_LIB

- partout : super pour ton moteur
- Apple platforms : `xcrun --sdk ... clang++ ... ar ...` puis **XCFramework**.

Toolchain Apple “propre” pour ton système (ce que ton builder doit produire)

1) Toujours passer par `xcrun`

Parce que `xcrun` choisit le bon `clang`, le bon `ar`, et surtout le bon **SDK**.

Exemples :

- macOS : `xcrun --sdk macosx clang++ ...`
- iOS device : `xcrun --sdk iphoneos clang++ ...`
- iOS sim : `xcrun --sdk iphonesimulator clang++ ...`

2) Flags minimums à générer (Apple)

- Sysroot : `-isysroot $(xcrun --sdk XXX --show-sdk-path)`
- Arch : `-arch arm64` (et parfois x86_64 simulator)
- Minimum OS :
  - macOS : `-mmacosx-version-min=11.0`
  - iOS : `-miphoneos-version-min=13.0`

- iOS sim : `-mios-simulator-version-min=13.0`
- tvOS : `-mappletvos-version-min=...`
- tvOS sim : `-mtvos-simulator-version-min=...`
- watchOS : `-mwatchos-version-min=...`
- watchOS sim : `-mwatchos-simulator-version-min=...`

3) ObjC++ / Cocoa frameworks

- macOS windowed : `-framework Cocoa` (+ éventuellement QuartzCore/Metal)
- iOS/tvOS : `-framework UIKit` (+ Metal, QuartzCore…)

Exemple “toolchain apple_clang” dans ton DSL (concret)

Tu peux modéliser une toolchain par **SDK** + **platform** + **arch**.

- `apple_macos_arm64`
- `apple_ios_arm64`
- `apple_iossim_arm64`
- `apple_iossim_x86_64`
  etc.

Chaque toolchain fournit :

- `cxxPath = "xcrun --sdk iphoneos clang++"` (ou tu gardes `xcrun` comme “launcher”)
- `sysroot = xcrun --show-sdk-path`
- `targetTriple` optionnel (pas obligatoire si tu utilises `-arch` + sdk)

Recommandation clé pour ton build system

Au lieu de stocker `cxxPath="clang++"`, stocke :

- `compilerDriver = ["xcrun","--sdk","iphoneos","clang++"]`
  ou un champ `useXcrunSdk="iphoneos"` + `cxx="clang++"`.

Ça t’évite 1000 bugs.

Pour tes fichiers `.mm` : où c’est “bon” et où faut faire attention

☑ Bon :

- `main.mm` / `AppDelegate.mm` / `Window.mm` / `View.mm`
- capture events OS → callbacks C++

⚠ À éviter :

- inclure `<Cocoa/Cocoa.h>` dans des headers utilisés par le core C++ (ça pollue tout)
- exposer des types ObjC dans l’API publique C++

Astuce : dans les headers, utilise **PIMPL** :

```
struct WindowImpl;
class Window { WindowImpl* _impl; };
```

Et maintenant, le plus utile pour toi

Si tu veux, je te fournis un “backend Apple” pour Jenga en pseudo-code qui génère **exactement** :

- compile `.cpp` / `.mm` → `.o`
- link consoleapp / macOS windowed
- build staticlib `.a`
- build sharedlib `.dylib`
- build XCFramework à partir de plusieurs `.a`

Dis-moi juste :

1. Ton Mac VM est **Intel ou Apple Silicon** ?
2. Tu veux supporter **x86_64 simulator** ou seulement **arm64 simulator** ?

Je te donne ensuite les templates de commandes finales (format “liste de strings” comme ton générateur).