**Documentation WebSocket**

**Didacticiel d'exemple d'application de CLIENT utilitaire**

**Chapitre 1 : Configuration initiale et configuration initiale Les bases**

Configuration des types de base, ouverture et fermeture de connexions, envoi et réception de messages.

**Étape 1**

Une boucle de programme de base qui invite l'utilisateur à saisir une commande, puis la traite. Dans ce didacticiel, nous modifierons ce programme pour effectuer des tâches et récupérer des données d'un serveur distant via une connexion WebSocket.

**Construire**

clang++ step1.cpp

**Code jusqu'à présent**

*remarque* Un instantané de code pour chaque étape est présent à côté de ce fichier de didacticiel dans le référentiel git.

#include <iostream>

#include <chaîne>

int main() {

bool done = false;

std :: entrée de chaîne ;

pendant (!done) {

std::cout << "Entrer la commande : ";

std::getline(std::cin, entrée);

if (input == "quitter") {

done = true;

} else if (input == " aide") {

std :: cout

<< "\nListe de commandes :\n"

<< "aide : afficher ce texte d'aide\n"

<< "quitter : quitter le programme\n"

≪≪ std::endl;

} autre {

std::cout << "Commande non reconnue" << std::endl;

}

}

retour 0;

}

**Étape 2**

*Ajoutez des inclusions WebSocket++ et configurez un type de point de terminaison.*

WebSocket++ comprend deux types d'objets principaux. Le point final et la connexion. Le point de terminaison crée et lance de nouvelles connexions et conserve les paramètres par défaut pour ces connexions. Les points de terminaison gèrent également toutes les ressources réseau partagées.

La connexion stocke les informations spécifiques à chaque session WebSocket.

**Remarque :** Une fois qu'une connexion est lancée, il n'y a aucun lien entre le point de terminaison et la connexion. Tous les paramètres par défaut sont copiés dans la nouvelle connexion par le point de terminaison. La modification des paramètres par défaut sur un point de terminaison n’affectera que les connexions futures.

Les connexions ne maintiennent pas de lien vers leur point de terminaison associé. Les points de terminaison ne conservent pas de liste des connexions en attente. Si votre application doit parcourir toutes les connexions, elle devra en conserver elle-même la liste.

Les points de terminaison WebSocket++ sont créés en combinant un rôle de point de terminaison avec une configuration de point de terminaison. Il existe deux types différents de rôles de point de terminaison, un pour les rôles client et un pour serveur dans une session WebSocket. Il s'agit d'un didacticiel client, nous allons donc utiliser le rôle client **[websocketpp::client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html" \o "Client endpoint role based on the given config.)** qui est fourni par l'en-tête .<**[websocketpp/client.hpp](https://docs.websocketpp.org/client_8hpp_source.html)**>

**Terminologie : configuration du point de terminaison**

Les points de terminaison WebSocket++ disposent d'un groupe de paramètres qui peuvent être configurés au moment de la compilation via le paramètre de modèle config. Une configuration est une structure qui contient des types et des constantes statiques utilisées pour produire un point de terminaison avec des propriétés spécifiques. En fonction de la configuration utilisée, le point de terminaison aura différentes méthodes disponibles et pourra avoir des dépendances tierces supplémentaires.

Le rôle de point de terminaison prend un paramètre de modèle appelé config qui est utilisé pour configurer le comportement du point de terminaison au moment de la compilation. Pour cet exemple, nous allons utiliser une configuration par défaut fournie par la bibliothèque appelée asio\_client, fournie par . Il s'agit d'une configuration client qui utilise boost::asio pour fournir le transport réseau et ne prend pas en charge la sécurité basée sur TLS. Plus tard, nous verrons comment introduire la sécurité basée sur TLS dans une application WebSocket++, plus sur les autres configurations de stock et comment créer vos propres configurations personnalisées.<**[websocketpp/config/asio\_no\_tls\_client.hpp](https://docs.websocketpp.org/asio__no__tls__client_8hpp_source.html)**>

Combinez une configuration avec un rôle de point de terminaison pour produire un point de terminaison entièrement configuré. Ce type sera fréquemment utilisé, je recommanderais donc un typedef ici.

typedef **[websocketpp::client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html" \o "Client endpoint role based on the given config.)**<**[websocketpp::config::asio\_client](https://docs.websocketpp.org/structwebsocketpp_1_1config_1_1asio__client.html" \o "Client config with asio transport and TLS disabled.)**> client

**Construire**

L'ajout de WebSocket++ a ajouté quelques dépendances à notre programme qui doivent être traitées dans le système de build. Premièrement, les en-têtes des bibliothèques WebSocket++ et Boost doivent se trouver dans le chemin de recherche d'inclusion de votre système de build. La manière exacte dont cela est effectué dépend de l'endroit où vous avez installé les en-têtes WebSocket++ et du système de build que vous utilisez.

En plus des nouveaux en-têtes, boost::asio dépend de la boost\_system bibliothèque partagée. Cela devra être ajouté (soit en tant que statique ou dynamique) à l'éditeur de liens. Reportez-vous à la documentation de votre environnement de build pour obtenir des instructions sur la création de liens vers des bibliothèques partagées.

clang++ step2.cpp -lboost\_system

**Code jusqu'à présent**

#include <websocketpp/config/asio\_no\_tls\_client.hpp>

#include <websocketpp/client.hpp>

#include <iostream>

#include <chaîne>

typedef [websocketpp::client<websocketpp::config::asio\_client>](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) [client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html);

int main() {

bool done = false;

std :: entrée de chaîne ;

pendant (!done) {

std::cout << "Entrer la commande : ";

std::getline(std::cin, entrée);

if (input == "quitter") {

done = true;

} else if (input == " aide") {

std :: cout

<< "\nListe de commandes :\n"

<< "aide : afficher ce texte d'aide\n"

<< "quitter : quitter le programme\n"

≪≪ std::endl;

} autre {

std::cout << "Commande non reconnue" << std::endl;

}

}

retour 0;

}

**Étape 3**

*Créez un objet wrapper de point de terminaison qui gère l’initialisation et la configuration du thread d’arrière-plan.*

Afin de traiter les entrées de l'utilisateur pendant que le traitement réseau s'effectue en arrière-plan, nous allons utiliser un thread distinct pour la boucle de traitement WebSocket++. Cela laisse le thread principal libre de traiter les entrées utilisateur au premier plan. Afin de permettre une gestion simple des ressources de style RAII pour notre thread et notre point de terminaison, nous utiliserons un objet wrapper qui les configure tous les deux dans son constructeur.

**Terminologie : espace de noms websocketpp::lib**

WebSocket++ est conçu pour être utilisé avec une bibliothèque standard C++11. Comme cela n'est pas universellement disponible dans les systèmes de build populaires, les bibliothèques Boost peuvent être utilisées comme polyfills pour la bibliothèque standard C++11 dans les environnements de build C++98. L'espace de noms websocketpp::lib est utilisé par la bibliothèque et ses exemples associés pour faire abstraction des distinctions entre les deux. websocketpp::lib::shared\_ptr sera évalué à std::shared\_ptr dans un environnement C++11 et boost::shared\_ptr dans le cas contraire.

Ce tutoriel utilise les websocketpp::lib wrappers car il ne sait pas quel est l'environnement de construction du lecteur. Pour vos applications, à moins que vous ne soyez intéressé par une portabilité similaire, vous êtes libre d'utiliser directement les versions boost ou std de ces types.

>[TODO : lien vers plus d'informations sur l'espace de noms websocketpp::lib et la configuration C++11]

Au sein du constructeur **[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html)** plusieurs choses se produisent :

Tout d’abord, nous définissons le comportement de journalisation des points de terminaison sur silencieux en effaçant tous les canaux d’accès et de journalisation des erreurs. [TODO : lien vers plus d'informations sur la journalisation]

m\_endpoint.clear\_access\_channels([websocketpp::log::alevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html" \l "a853aa0b8976e53f3181af3bc398d493e));

m\_endpoint.clear\_error\_channels([websocketpp::log::elevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1elevel.html" \l "a9b31ff708c221d314f9f4eb3ff2b1ad7));

Ensuite, nous initialisons le système de transport sous-jacent au point de terminaison et le mettons en mode perpétuel. En mode perpétuel, la boucle de traitement du point de terminaison ne se terminera pas automatiquement lorsqu'il n'a aucune connexion. Ceci est important car nous voulons que ce point de terminaison reste actif pendant que notre application est en cours d'exécution et traite les demandes de nouvelles connexions WebSocket à la demande lorsque nous en avons besoin. Ces deux méthodes sont spécifiques au transport asio. Ils ne seront pas nécessaires ni présents dans les points de terminaison qui utilisent une configuration non-asio.

m\_endpoint.init\_asio();

m\_endpoint.start\_perpetual();

Enfin, nous lançons un thread pour exécuter la méthode run de notre point de terminaison client. Pendant que le point de terminaison est en cours d'exécution, il traitera les tâches de connexion (lire et transmettre les messages entrants, encadrer et envoyer des messages sortants, etc.). Parce qu'il fonctionne en mode perpétuel, lorsqu'aucune connexion n'est active, il attendra une nouvelle connexion.

m\_thread.reset(nouveau websocketpp::lib::thread(&client::run, &m\_endpoint));< /span>

**Construire**

Maintenant que notre modèle de point de terminaison client est réellement instancié, quelques dépendances supplémentaires de l'éditeur de liens apparaîtront. En particulier, les clients WebSocket nécessitent un générateur de nombres aléatoires cryptographiquement sécurisé. WebSocket++ est capable d'utiliser boost\_random ou la bibliothèque standard C++11 <random> dans ce but. Étant donné que cet exemple utilise également des threads, si nous n'avons pas C++11 std::thread disponible, nous devrons inclure boost\_thread.

**Clang (C++98 et boost)**

clang++ [**step3.cpp**](https://docs.websocketpp.org/step3_8cpp_source.html) -lboost\_system -lboost\_random -lboost\_thread

**Clang (C++11)**

clang++ -std=c++0x -stdlib=libc++ [**step3.cpp**](https://docs.websocketpp.org/step3_8cpp_source.html) -lboost\_system -D\_WEBSOCKETPP\_CPP11\_STL\_

**G++ (C++98 et Boost)**

g++ [**step3.cpp**](https://docs.websocketpp.org/step3_8cpp_source.html) -lboost\_system -lboost\_random -lboost\_thread

**G++ v4.6+ (C++11)**

g++ -std=c++0x [**step3.cpp**](https://docs.websocketpp.org/step3_8cpp_source.html) -lboost\_system -D\_WEBSOCKETPP\_CPP11\_STL\_

**Code jusqu'à présent**

#include <websocketpp/config/asio\_no\_tls\_client.hpp>

#include <websocketpp/client.hpp>

#include <websocketpp/common/thread.hpp>

#include <websocketpp/common/memory.hpp>

#include <iostream>

#include <chaîne>

typedef [websocketpp::client<websocketpp::config::asio\_client>](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) [client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html);

classe [websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) {

public :

[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) () {

m\_endpoint.[clear\_access\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a3fe357b3c6434273a9965eb52902575d)([websocketpp::log::alevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html" \l "a853aa0b8976e53f3181af3bc398d493e) );

m\_endpoint.[clear\_error\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a8cefa4fc37c0df4eb6ed0bcf04bb0da5)([websocketpp::log::elevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1elevel.html" \l "a9b31ff708c221d314f9f4eb3ff2b1ad7) );

m\_endpoint.init\_asio();

m\_endpoint.start\_perpetual();

m\_thread.reset(nouveau websocketpp::lib::thread(&client::run, &m\_endpoint));< /span>

}

privé :

[client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) m\_endpoint;

websocketpp::lib::shared\_ptr<websocketpp::lib::thread> m\_thread;

} ;

int main() {

bool done = false;

std :: entrée de chaîne ;

[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) point de terminaison ;

pendant (!done) {

std::cout << "Entrer la commande : ";

std::getline(std::cin, entrée);

if (input == "quitter") {

done = true;

} else if (input == " aide") {

std :: cout

<< "\nListe de commandes :\n"

<< "aide : afficher ce texte d'aide\n"

<< "quitter : quitter le programme\n"

≪≪ std::endl;

} autre {

std::cout << "Commande non reconnue" << std::endl;

}

}

retour 0;

}

**Étape 4**

*Ouverture des connexions WebSocket*

Cette étape ajoute deux nouvelles commandes à utility\_client. La possibilité d'ouvrir une nouvelle connexion et la possibilité d'afficher des informations sur une connexion précédemment ouverte. Chaque connexion ouverte se verra attribuer un identifiant de connexion entier que l'utilisateur du programme pourra utiliser pour interagir avec cette connexion.

**Nouvel objet de métadonnées de connexion**

Afin de suivre les informations sur chaque connexion, un objet **[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)** est défini. Cet objet stocke l'identifiant de connexion numérique et un certain nombre de champs qui seront remplis au fur et à mesure du traitement de la connexion. Initialement, cela inclut l'état de la connexion (ouverture, ouverte, échec, fermée, etc.), l'URI d'origine auquel la connexion est connectée, une valeur d'identification du serveur et une description de la raison de l'échec/fermeture de la connexion. Les étapes suivantes ajouteront plus d'informations à cet objet de métadonnées.

**Mettre à jour <tt>websocket\_endpoint</tt>**

L'objet **[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html)** a gagné de nouveaux membres de données et méthodes. Il suit désormais un mappage entre les identifiants de connexion et leurs métadonnées associées ainsi que le prochain numéro d'identification séquentiel à distribuer. La méthode connect() initie une nouvelle connexion. La méthode get\_metadata récupère les métadonnées en fonction d'un identifiant.

**La méthode de connexion**

Une nouvelle connexion WebSocket est initiée via un processus en trois étapes. Tout d'abord, une demande de connexion est créée par endpoint::get\_connection(uri). Ensuite, la demande de connexion est configurée. Enfin, la demande de connexion est renvoyée au point de terminaison via endpoint::connect() qui l'ajoute à la file d'attente des nouvelles connexions à établir.

**Terminologie <tt>connection\_ptr</tt>**

WebSocket++ assure le suivi des ressources liées à la connexion à l'aide d'un pointeur partagé compté par référence. Le type de ce pointeur est endpoint::connection\_ptr. Un connection\_ptr permet d'accéder directement aux informations sur la connexion et permet de modifier les paramètres de connexion. En raison de cet accès direct et de leur rôle de gestion des ressources internes au sein de la bibliothèque, il n'est pas sûr pour les applications finales d'utiliser connection\_ptr, sauf dans les circonstances spécifiques détaillées ci-dessous.

**Quand est-il sécuritaire d'utiliser connection\_ptr ?**

* Après endpoint::get\_connection(...) et avant endpoint::connect() : get\_connection renvoie un connection\_ptr. Vous pouvez utiliser ce pointeur en toute sécurité pour configurer votre nouvelle connexion. Une fois que vous avez soumis la connexion à connect, vous ne pouvez plus utiliser le connection\_ptr et devez le supprimer immédiatement pour une gestion optimale de la mémoire.
* Pendant un gestionnaire : WebSocket++ vous permet d'enregistrer des hooks/rappels/gestionnaires d'événements pour des événements spécifiques qui se produisent pendant la durée de vie d'une connexion. Lors de l'invocation de l'un de ces gestionnaires, la bibliothèque garantit qu'il est sûr d'utiliser un connection\_ptr pour la connexion associée au gestionnaire en cours d'exécution.

**Terminologie <tt>connection\_hdl</tt>**

En raison de la sécurité limitée des threads du connection\_ptr, la bibliothèque fournit également un identifiant de connexion plus flexible, le connection\_hdl. Le connection\_hdl a le type **[websocketpp::connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html" \l "a6b3d26a10ee7229b84b776786332631d" \o "A handle to uniquely identify a connection.)** et il est défini dans . Notez que contrairement à cela ne dépend pas du type ou de la configuration du point de terminaison. Le code qui stocke ou transmet simplement mais ne les utilise pas peut inclure uniquement l'en-tête ci-dessus et peut traiter ses hdls comme des valeurs.<**[websocketpp/common/connection\_hdl.hpp](https://docs.websocketpp.org/connection__hdl_8hpp_source.html)**>connection\_ptrconnection\_hdl

Les handles de connexion ne sont pas utilisés directement. Ils sont utilisés par les méthodes de point final pour identifier la cible de l'action souhaitée. Par exemple, la méthode de point de terminaison qui envoie un nouveau message prendra comme paramètre le hdl de la connexion à laquelle envoyer le message.

**Quand est-il sûr d'utiliser connection\_hdl ?** connection\_hdls peuvent être utilisés à tout moment à partir de n'importe quel fil de discussion. Ils peuvent être copiés et stockés dans des conteneurs. La suppression d'un hdl n'affectera en aucun cas la connexion. Les handles peuvent être mis à niveau vers un connection\_ptr lors d'un appel de handler en utilisant endpoint::get\_con\_from\_hdl(). Le connection\_ptr résultant peut être utilisé en toute sécurité pendant la durée de l'invocation de ce gestionnaire.

\*\*connection\_hdl FAQ\*\*

* connection\_hdlLes s sont garantis uniques au sein d’un programme. Plusieurs points de terminaison dans un seul programme créeront toujours des connexions avec des identifiants uniques.
* L'utilisation d'un connection\_hdl avec un point de terminaison différent de celui qui a créé sa connexion associée entraînera un comportement indéfini.
* Utiliser un connection\_hdl dont la connexion associée a été fermée ou supprimée est sécurisé. Le point de terminaison renverra une erreur spécifique indiquant que l'opération n'a pas pu être terminée car la connexion associée n'existe pas. [À FAIRE : plus ici ? un lien vers une FAQ connection\_hdl ailleurs ?]

websocket\_endpoint::connect() commence par appeler endpoint::get\_connection() en utilisant un uri passé en paramètre. De plus, une valeur de sortie d'erreur est transmise pour capturer toutes les erreurs qui pourraient survenir. Si une erreur se produit, un avis d'erreur est imprimé avec un message descriptif et le symbole -1 / « invalide ». la valeur est renvoyée comme nouvel identifiant.

**Terminologie:error handling: exceptions vs error\_code**

WebSocket++ utilise le système de codes d'erreur défini par la bibliothèque C++11 <system\_error>. Il peut éventuellement recourir à un système similaire fourni par les bibliothèques Boost. Toutes les méthodes de point de terminaison destinées aux utilisateurs qui peuvent échouer prennent un error\_code dans un paramètre de sortie et stockent l'erreur qui s'y est produite avant de revenir. Une valeur construite vide/par défaut est renvoyée en cas de succès.

**Variantes de génération d'exceptions** Toutes les méthodes de point de terminaison destinées aux utilisateurs qui prennent et utilisent un paramètre error\_code ont une version qui lève une exception à la place. Ces méthodes sont identiques en fonction et en signature, à l'exception de l'absence du paramètre ec final. Le type de l'exception levée est **[websocketpp::exception](https://docs.websocketpp.org/classwebsocketpp_1_1exception.html)**. Ce type dérive de std::exception il peut donc être intercepté par des blocs catch saisissant des std::exception génériques. La méthode websocketpp::exception::code() peut être utilisée pour extraire la valeur lisible par machine error\_code d'une exception.

Pour plus de clarté sur la gestion des erreurs, l'exemple utility\_client utilise exclusivement les variantes sans exception de ces méthodes. Votre application peut choisir d’utiliser l’un ou l’autre.

Si la création de la connexion réussit, l'ID de connexion séquentielle suivant est généré et un **[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)** objet est inséré dans la liste de connexions sous cet ID. Initialement, l'objet de métadonnées stocke l'ID de connexion, le connection\_hdl et l'URI auquel la connexion a été ouverte.

int new\_id = m\_next\_id++;

metadata\_ptr metadata(new [connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)(new\_id, con->get\_handle(), uri) );

m\_connection\_list[new\_id] = métadonnées ;

Ensuite, la demande de connexion est configurée. Pour cette étape, la seule configuration que nous ferons consiste à configurer quelques gestionnaires par défaut. Plus tard, nous reviendrons et démontrerons une configuration plus détaillée qui peut se produire ici (définition des agents utilisateurs, de l'origine, des proxys, des en-têtes personnalisés, des sous-protocoles, etc.).

**Terminologie : enregistrement des gestionnaires**

WebSocket++ fournit un certain nombre de points d'exécution où vous pouvez vous inscrire pour exécuter un gestionnaire. Les points disponibles pour votre point de terminaison dépendront de sa configuration. Les gestionnaires TLS n'existeront pas sur les points de terminaison non TLS par exemple. Une liste complète des gestionnaires est disponible sur <http://www.zaphoyd.com/websocketpp/manual/reference/handler-list>.

Les gestionnaires peuvent être enregistrés au niveau du point de terminaison et au niveau de la connexion. Les gestionnaires de points de terminaison sont copiés dans les nouvelles connexions au fur et à mesure de leur création. La modification d’un gestionnaire de point de terminaison affectera uniquement les connexions futures. Les gestionnaires enregistrés au niveau de la connexion seront liés uniquement à cette connexion spécifique.

La signature des méthodes de liaison du gestionnaire est la même pour les points de terminaison et les connexions. Le format est : set\_\*\_handler(...). Où \* est le nom du gestionnaire. Par exemple, set\_open\_handler(...) définira le gestionnaire à appeler lorsqu'une nouvelle connexion est ouverte. set\_fail\_handler(...) définira le gestionnaire à appeler lorsqu'une connexion échoue.

Tous les gestionnaires prennent un argument, un type appelable qui peut être converti en std::function avec le nombre et le type d'arguments corrects. Vous pouvez transmettre des fonctions gratuites, des foncteurs et des Lambdas avec des listes d'arguments correspondantes en tant que gestionnaires. De plus, vous pouvez utiliser std::bind (ou boost::bind) pour enregistrer des fonctions avec des listes d'arguments qui ne correspondent pas. Ceci est utile pour transmettre des paramètres supplémentaires non présents dans la signature du gestionnaire ou dans les fonctions membres qui doivent porter un « ce » ; pointeur.

La signature de fonction de chaque gestionnaire peut être recherchée dans la liste ci-dessus dans le manuel. En général, tous les gestionnaires incluent le connection\_hdl identifiant la connexion à laquelle il est associé comme premier paramètre. Certains gestionnaires (tels que le gestionnaire de messages) incluent des paramètres supplémentaires. La plupart des gestionnaires ont une valeur de retour vide, mais certains (validate, ping, tls\_init) n'en ont pas. Les significations spécifiques des valeurs de retour sont documentées dans la liste des gestionnaires liée ci-dessus.

utility\_clientenregistre un gestionnaire d'ouverture et d'échec. Nous les utiliserons pour savoir si chaque connexion a été ouverte avec succès ou a échoué. S'il s'ouvre avec succès, nous rassemblerons certaines informations de la poignée de main d'ouverture et les stockerons avec nos métadonnées de connexion.

Dans cet exemple, nous allons définir des gestionnaires spécifiques à la connexion qui sont directement liés à l'objet de métadonnées associé à notre connexion. Cela nous permet d'éviter d'effectuer une recherche dans chaque gestionnaire pour trouver l'objet de métadonnées que nous prévoyons de mettre à jour, ce qui est un peu plus efficace.

Examinons en détail les paramètres envoyés pour la liaison :

con->set\_open\_handler(websocketpp::lib::bind(

&connection\_metadata::on\_open,

métadonnées,

&m\_endpoint,

websocketpp::lib::placeholders::\_1

));

&connection\_metadata::on\_open est l'adresse de la fonction membre on\_open de la classe **[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)**. metadata\_ptr est un pointeur vers l'objet **[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)** associé à cette classe. Il sera utilisé comme objet sur lequel la fonction membre on\_open sera appelée. &m\_endpoint est l'adresse du point de terminaison utilisé. Ce paramètre sera passé tel quel à la méthode on\_open. Enfin, websocketpp::lib::placeholders::\_1 est un espace réservé indiquant que la fonction liée doit prendre un argument supplémentaire à remplir ultérieurement. WebSocket++ remplira cet espace réservé avec connection\_hdl lorsqu'il invoquera le gestionnaire.

Enfin, nous appelons endpoint::connect() sur notre demande de connexion configurée et renvoyons le nouvel ID de connexion.

**Fonctions des membres du gestionnaire**

Le gestionnaire d'ouverture que nous avons enregistré, connection\_metadata::on\_open, définit le champ de métadonnées d'état sur « Ouvert » et récupère la valeur du "Serveur" en-tête de la réponse HTTP du point de terminaison distant et le stocke dans l'objet de métadonnées. Les serveurs définissent souvent une chaîne d'identification dans cet en-tête.

Le gestionnaire d'échec que nous avons enregistré, connection\_metadata::on\_fail, définit le champ de métadonnées d'état sur "Échec", le champ du serveur de la même manière que on\_open, et récupère le code d'erreur décrivant la raison pour laquelle la connexion a échoué. Le message lisible par l'homme associé à ce code d'erreur est enregistré dans l'objet de métadonnées.

**Nouvelles commandes**

Deux nouvelles commandes ont été configurées. "connecter [uri]" transmettra l'URI à la méthode de connexion **[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html)** et signalera une erreur ou l'ID de connexion de la nouvelle connexion. "afficher [identifiant de connexion]" récupérera et imprimera les métadonnées associées à cette connexion. Le texte d'aide a été mis à jour en conséquence.

} else if (input.substr(0,7) == "connecter") {

int id = endpoint.connect(input.substr(8));

if (id != -1) {

std::cout << "> Connexion créée avec l'identifiant " << identifiant << std::endl;

}

} else if (input.substr(0,4) == "afficher") {

int id = atoi(input.substr(5).c\_str());

connection\_metadata::ptr metadata = endpoint.get\_metadata(id);

if (métadonnées) {

std :: cout << \*métadonnées << std::endl;

} autre {

std::cout << "> ID de connexion inconnu " << identifiant << std::endl;

}

}

**Construire**

Il n'y a aucun changement dans les instructions de construction depuis l'étape 3

**Courir**

Entrez la commande : ne connectez pas un uri de websocket

> Erreur d'initialisation de connexion : uri invalide

Entrez la commande : affichez 0

> ID de connexion inconnu 0

Entrez la commande : connectez ws://echo.websocket.org

> Connexion créée avec l'identifiant 0

Entrez la commande : affichez 0

> URI : ws://echo.websocket.org

> Statut : Ouvert

> Serveur distant : passerelle Kaazing

> Erreur/raison de fermeture : N/A

Entrez la commande : connectez-vous à ws://wikipedia.org

> Connexion créée avec l'identifiant 1

Entrez la commande : afficher 1

> Vous êtes sur : ws://wikipedia.org

> Statut : Échec

> Serveur distant : Apache

> Raison de l'erreur/fermeture : statut HTTP non valide.

**Code jusqu'à présent**

#include <websocketpp/config/asio\_no\_tls\_client.hpp>

#include <websocketpp/client.hpp>

#include <websocketpp/common/thread.hpp>

#include <websocketpp/common/memory.hpp>

#include <cstdlib>

#include <iostream>

#include <carte>

#include <chaîne>

#include <sstream>

typedef [websocketpp::client<websocketpp::config::asio\_client>](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) [client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html);

classe [connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html) {

public :

typedef websocketpp::lib::shared\_ptr<connection\_metadata> ptr;

[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)(int id, [websocketpp::connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html#a6b3d26a10ee7229b84b776786332631d) hdl, std::string uri)

: m\_id(id)

, m\_hdl(hdl)

, m\_status("Connexion")

, m\_u(s)

, m\_server("N/A")

{}

void on\_open([client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) \* c, [websocketpp : :connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html#a6b3d26a10ee7229b84b776786332631d) hdl) {

m\_status = "Ouvrir";

[client::connection\_ptr](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#a2e187bbb2beac676bbfbc2e8065de83e) con = c->[get\_con\_from\_hdl](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html" \l "a0fe4457427d4124abe7ca022ba7afbb4)(hdl);

m\_server = con->get\_response\_header("Serveur");

}

void on\_fail([client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) \* c, [websocketpp : :connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html#a6b3d26a10ee7229b84b776786332631d) hdl) {

m\_status = "Échec";

[client::connection\_ptr](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#a2e187bbb2beac676bbfbc2e8065de83e) con = c->[get\_con\_from\_hdl](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html" \l "a0fe4457427d4124abe7ca022ba7afbb4)(hdl);

m\_server = con->get\_response\_header("Serveur");

m\_error\_reason = con->get\_ec().message();

}

ami std::ostream & opérateur≪≪ (std::ostream & out, [connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html) const & data) ;

privé :

int m\_id;

[websocketpp::connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html#a6b3d26a10ee7229b84b776786332631d) m\_hdl;

std :: chaîne m\_status ;

std :: chaîne m\_uri ;

std :: chaîne m\_server ;

std::string m\_error\_reason;

} ;

std::ostream & opérateur≪≪ (std::ostream & out, [connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html) const & data) {

sortie << "> URI : " << data.m\_uri ≪≪ "\n"

<< "> Statut : " << data.m\_status << "\n"

<< "> Serveur distant : " << (data.m\_server.empty() ? "Aucun spécifié" : data.m\_server) << "\n"

<< "> Erreur/raison de fermeture : " << (data.m\_error\_reason.empty() ? "N/A" : data.m\_error\_reason);

retour sortie ;

}

classe [websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) {

public :

[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) () : m\_next\_id(0) {

m\_endpoint.[clear\_access\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a3fe357b3c6434273a9965eb52902575d)([websocketpp::log::alevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html" \l "a853aa0b8976e53f3181af3bc398d493e) );

m\_endpoint.[clear\_error\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a8cefa4fc37c0df4eb6ed0bcf04bb0da5)([websocketpp::log::elevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1elevel.html" \l "a9b31ff708c221d314f9f4eb3ff2b1ad7) );

m\_endpoint.init\_asio();

m\_endpoint.start\_perpetual();

m\_thread.reset(nouveau websocketpp::lib::thread(&client::run, &m\_endpoint));< /span>

}

int connect(std::string const & uri) {

websocketpp::lib::error\_code ec;

[client::connection\_ptr](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#a2e187bbb2beac676bbfbc2e8065de83e) con = m\_endpoint.[get\_connection](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#aa95cec974921e32bc37a61c683ae240d)(uri, ec);

si (ec) {

std::cout << "> Erreur d'initialisation de connexion : " << ec.message() << std::endl;

retour -1;

}

int new\_id = m\_next\_id++;

connection\_metadata::ptr metadata\_ptr(new [connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)(new\_id, con->get\_handle() , uri));

m\_connection\_list[new\_id] = metadata\_ptr;

con->set\_open\_handler(websocketpp::lib::bind(

&connection\_metadata::on\_open,

métadonnées\_ptr,

&m\_endpoint,

websocketpp::lib::placeholders::\_1

));

con->set\_fail\_handler(websocketpp::lib::bind(

&connection\_metadata::on\_fail,

métadonnées\_ptr,

&m\_endpoint,

websocketpp::lib::placeholders::\_1

));

m\_endpoint.[connecter](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#a818c30343180123bf1fee6dc21524bae)(con);

retour new\_id;

}

connection\_metadata::ptr get\_metadata(int id) const {

con\_list::const\_iterator metadata\_it = m\_connection\_list.find(id);

if (metadata\_it == m\_connection\_list.end()) {

retour connection\_metadata::ptr();

} autre {

retour metadata\_it->seconde;

}

}

privé :

typedef std::map<int,connection\_metadata::ptr> con\_list;

[client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) m\_endpoint;

websocketpp::lib::shared\_ptr<websocketpp::lib::thread> m\_thread;

con\_list m\_connection\_list;

int m\_next\_id;

} ;

int main() {

bool done = false;

std :: entrée de chaîne ;

[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html) point de terminaison ;

pendant (!done) {

std::cout << "Entrer la commande : ";

std::getline(std::cin, entrée);

if (input == "quitter") {

done = true;

} else if (input == " aide") {

std :: cout

<< "\nListe de commandes :\n"

<< "connecter <ws uri>\n"

<< "afficher <identifiant de connexion>\n"

<< "aide : afficher ce texte d'aide\n"

<< "quitter : quitter le programme\n"

≪≪ std::endl;

} else if (input.substr(0,7) == "connecter") {

int id = endpoint.connect(input.substr(8));

if (id != -1) {

std::cout << "> Connexion créée avec l'identifiant " << identifiant << std::endl;

}

} else if (input.substr(0,4) == "afficher") {

int id = atoi(input.substr(5).c\_str());

connection\_metadata::ptr metadata = endpoint.get\_metadata(id);

if (métadonnées) {

std :: cout << \*métadonnées << std::endl;

} autre {

std::cout << "> ID de connexion inconnu " << identifiant << std::endl;

}

} autre {

std::cout << "> Commande non reconnue" << std::endl;

}

}

retour 0;

}

**Étape 5**

*Fermeture des connexions*

Cette étape ajoute une commande qui vous permet de fermer une connexion WebSocket et ajuste la commande quit afin qu'elle ferme proprement toutes les connexions en attente avant de quitter.

**Obtenir des informations de fermeture de connexion à partir de WebSocket++**

**Terminologie : codes de fermeture WebSocket et amp; les raisons**

La négociation de fermeture WebSocket implique un échange de codes de fermeture facultatifs lisibles par une machine et de chaînes de raison lisibles par l'homme. Chaque point de terminaison envoie des détails de clôture indépendants. Les codes sont des entiers courts. Les raisons sont des chaînes de texte UTF8 d'au plus 125 caractères. Plus de détails sur les plages de codes de fermeture valides et la signification de chaque code sont disponibles sur <https://tools.ietf.org/html/rfc6455#section-7.4>

L'espace de noms **[websocketpp::close::status](https://docs.websocketpp.org/namespacewebsocketpp_1_1close_1_1status.html" \o "A package of types and methods for manipulating WebSocket close status'.)** contient des constantes nommées pour tous les codes de fermeture définis par l'IANA. Il comprend également des fonctions gratuites pour déterminer si une valeur est réservée ou invalide et pour convertir un code en une représentation textuelle lisible par l'homme.

Pendant l'appel du gestionnaire de fermeture, les connexions WebSocket++ offrent les méthodes suivantes pour accéder aux informations de négociation de fermeture :

* connection::get\_remote\_close\_code() : Obtenez le code de fermeture tel que rapporté par le point de terminaison distant
* connection::get\_remote\_close\_reason() : Obtenez la raison de la fermeture telle que rapportée par le point de terminaison distant
* connection::get\_local\_close\_code() : obtenez le code de fermeture envoyé par ce point de terminaison.
* connection::get\_local\_close\_reason() : obtenez la raison proche pour laquelle ce point de terminaison a été envoyé.
* connection::get\_ec() : obtenez un WebSocket++ error\_code plus détaillé/spécifique indiquant quelle erreur de bibliothèque (le cas échéant) a finalement entraîné la fermeture de la connexion.

*Remarque :* il existe des codes de fermeture spéciaux qui signaleront un code qui n'a pas été réellement envoyé sur le fil. Par exemple 1005/"pas de code de fermeture" indique que le point de terminaison a entièrement omis un code de fermeture et que 1006/"fermeture anormale" indique qu'un problème s'est produit qui a entraîné la fermeture de la connexion sans avoir effectué une négociation de fermeture.

**Ajouter un gestionnaire de fermeture**

La méthode connection\_metadata::on\_close est ajoutée. Cette méthode récupère le code de fermeture et le motif de la négociation de clôture et les stocke dans le champ de motif d'erreur local.

void on\_close([client](https://docs.websocketpp.org/classwebsocketpp_1_1client.html) \* c, [websocketpp : :connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html#a6b3d26a10ee7229b84b776786332631d) hdl) {

m\_status = "Fermé";

[client::connection\_ptr](https://docs.websocketpp.org/classwebsocketpp_1_1client.html#a2e187bbb2beac676bbfbc2e8065de83e) con = c->[get\_con\_from\_hdl](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html" \l "a0fe4457427d4124abe7ca022ba7afbb4)(hdl);

std :: stringstream s ;

s << "code de fermeture : " << con->get\_remote\_close\_code() << " ("

<< [websocketpp::close::status::get\_string](https://docs.websocketpp.org/namespacewebsocketpp_1_1close_1_1status.html#a4bf4987c79165b134ed1207a567ce209)(con->get\_remote\_close\_code())

<< "), motif de fermeture : " << con->get\_remote\_close\_reason();

m\_error\_reason = s.str();

}

De la même manière que on\_open et on\_fail, websocket\_endpoint::connect enregistre ce gestionnaire de fermeture lorsqu'une nouvelle connexion est établie.

**Ajoutez la méthode close à <tt>websocket\_endpoint</tt>**

Cette méthode commence par rechercher l'ID de connexion donné dans la liste des connexions. Ensuite, une demande de fermeture est envoyée au handle de la connexion avec le code de fermeture WebSocket spécifié. Cela se fait en appelant endpoint::close. Il s'agit d'une méthode thread-safe utilisée pour envoyer de manière asynchrone un signal de fermeture à la connexion avec le handle donné. Une fois l'opération terminée, le gestionnaire de fermeture de la connexion sera déclenché.

void close(int id, code) {[websocketpp::close::status::value](https://docs.websocketpp.org/namespacewebsocketpp_1_1close_1_1status.html" \l "a8614a5c4733d708e2d2a32191c5bef84)

websocketpp::lib::error\_code ec;

con\_list::iterator metadata\_it = m\_connection\_list.find(id);

if (metadata\_it == m\_connection\_list.end()) {

std::cout << "> Aucune connexion trouvée avec l'identifiant " << identifiant << std::endl;

retour;

}

m\_endpoint.close(metadata\_it->second->get\_hdl(), code, "", ec);< /span>

si (ec) {

std::cout << "> Erreur lors du lancement de la fermeture : " << ec.message() << std::endl;

}

}

**Ajouter une option de fermeture à la boucle de commande et au message d'aide**

Une option de fermeture est ajoutée à la boucle de commande. Il faut un identifiant de connexion et éventuellement un code de fermeture et un motif de fermeture. Si aucun code n'est spécifié, la valeur par défaut 1000/Normal est utilisée. Si aucune raison n’est précisée, aucune n’est envoyée. La méthode endpoint::close effectuera une vérification des erreurs et annulera la demande de fermeture si vous essayez d'envoyer un code non valide ou une raison avec un formatage UTF8 non valide. Les chaînes de raison de plus de 125 caractères seront tronquées.

Une entrée est également ajoutée au système d'aide pour décrire comment la nouvelle commande peut être utilisée.

else if (input.substr(0,5) == "close"< /span>) {

std :: stringstream ss (entrée);

std :: chaîne cmd ;

int identifiant ;

int close\_code = [websocketpp::close::status::normal](https://docs.websocketpp.org/namespacewebsocketpp_1_1close_1_1status.html#a6f258fb124459f35789c1a7a2ab220e3);

std::chaîne raison ;

ss >> cmd>> id >> code\_ferme ;

std::getline(ss,raison);

endpoint.close(id, close\_code, Reason);

}

**Fermez toutes les connexions en attente dans <tt>websocket\_endpoint</tt> destructeur**

Jusqu'à présent, la fermeture du programme laissait les connexions en suspens et le thread réseau WebSocket++ dans une impasse. Maintenant que nous disposons d’une méthode pour fermer les connexions, nous pouvons nettoyer cela correctement.

Le destructeur de **[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html)** arrête désormais le mode perpétuel (de sorte que le thread d'exécution se termine après la fermeture de la dernière connexion) et parcourt la liste des connexions ouvertes et demande une fermeture propre pour chacune. Enfin, le thread d'exécution est rejoint, ce qui oblige le programme à attendre que la connexion soit terminée.

~[websocket\_endpoint](https://docs.websocketpp.org/classwebsocket__endpoint.html)() {

m\_endpoint.stop\_perpetual();

pour (con\_list::const\_iterator it = m\_connection\_list.begin(); it != m\_connection\_list.end(); ++it) {

if (it->second->get\_status() != "Ouvrir") {

// Ferme uniquement les connexions ouvertes

continuer;

}

std::cout << "> Fermeture de la connexion " << it->second->get\_id() << std::endl;

websocketpp::lib::error\_code ec;

m\_endpoint.close(it->second->get\_hdl(), [websocketpp::close::status::going\_away](https://docs.websocketpp.org/namespacewebsocketpp_1_1close_1_1status.html#afb385550a4c7a718e159ba4f13e2d9f6), "", ec);

si (ec) {

std::cout << "> Erreur lors de la fermeture de la connexion " << it->second->get\_id() << " : "

≪≪ ec.message() << std::endl;

}

}

m\_thread->join();

}

**Construire**

Il n'y a aucun changement dans les instructions de construction depuis l'étape 4

**Courir**

Entrez la commande : connectez ws://localhost:9002

> Connexion créée avec l'identifiant 0

Entrez la commande : fermez 0 1001, exemple de message

Entrez la commande : affichez 0

> URI : ws://localhost:9002

> Statut : Fermé

> Serveur distant : WebSocket++/0.4.0

> Erreur/raison de fermeture : code de fermeture : 1001 (Partir), raison de fermeture : exemple de message

Entrez la commande : connectez ws://localhost:9002

> Connexion créée avec l'identifiant 1

Entrez la commande : fermez 1 1006

> Erreur lors du lancement de la fermeture : code de fermeture non valide utilisé

Entrez la commande : quitter

> Fermeture de la connexion 1

**Étape 6**

*Envoi et réception de messages*

Cette étape ajoute une commande pour envoyer un message sur une connexion donnée et met à jour la commande show pour imprimer une transcription de tous les messages envoyés et reçus pour cette connexion.

**Terminologie : types de messages WebSocket (opcodes)**

Les messages WebSocket ont des types indiqués par leur opcode. Le protocole spécifie actuellement deux opcodes différents pour les messages de données, texte et binaire. Les messages texte représentent du texte UTF8 et seront validés comme tels. Les messages binaires représentent des octets binaires bruts et sont transmis directement sans validation.

WebSocket++ fournit les valeurs websocketpp::frame::opcode::text et websocketpp::frame::opcode::binary qui peuvent être utilisées pour indiquer la manière dont les messages sortants doivent être envoyés et pour vérifier le formatage des messages entrants.

**Envoi de messages**

Les messages sont envoyés à l'aide de endpoint::send. Il s'agit d'une méthode thread-safe qui peut être appelée de n'importe où pour mettre en file d'attente un message à envoyer sur la connexion spécifiée. Il existe trois surcharges d'envoi à utiliser avec différents scénarios.

Chaque méthode prend un connection\_hdl pour indiquer sur quelle connexion envoyer le message ainsi qu'un frame::opcode::value pour indiquer sous quel opcode étiqueter le message. Toutes les surcharges sont également disponibles avec une variante sans exception qui remplit un code d'état/d'erreur au lieu de le lancer.

La première surcharge, connection\_hdl hdl, std::string const & payload, frame::opcode::value op, prend un std::string. Le contenu de la chaîne est copié dans un tampon interne et peut être modifié en toute sécurité après avoir appelé send.

La deuxième surcharge, connection\_hdl hdl, void const \* payload, size\_t len, frame::opcode::value op, prend un tampon et une longueur void \*. Le contenu du tampon est copié et peut être modifié en toute sécurité après avoir appelé send.

La troisième surcharge, connection\_hdl hdl, message\_ptr msg, prend un WebSocket++ message\_ptr. Cette surcharge permet de construire un message sur place avant l'appel à envoyer. Cela peut également permettre d'envoyer un seul tampon de message plusieurs fois, y compris à plusieurs connexions, sans copie. Que cela se produise ou non dépend d'autres facteurs, tels que l'activation ou non de la compression. Le contenu du tampon de message ne peut pas être modifié en toute sécurité après son envoi.

**Terminologie : mise en file d'attente des messages WebSocket sortants et mise en file d'attente des messages WebSocket sortants. contrôle de flux**

Dans de nombreuses configurations, par exemple lorsque le transport basé sur Asio est utilisé, WebSocket++ est un système asynchrone. En tant que telle, la méthode endpoint::send peut être renvoyée avant que des octets ne soient réellement écrits sur le socket sortant. Dans les cas où send est appelé plusieurs fois de suite, les messages peuvent être fusionnés et envoyés dans la même opération ou même dans le même paquet TCP. Lorsque cela se produit, les limites du message sont préservées (chaque appel à envoyer produira un message distinct).

Dans le cas d'applications qui appellent send depuis un gestionnaire, cela signifie qu'aucun message ne sera écrit sur le socket jusqu'au retour de ce gestionnaire. Si vous prévoyez d'envoyer de nombreux messages dans ce manoir ou si vous avez besoin qu'un message soit écrit sur le fil avant de continuer, vous devriez envisager d'utiliser plusieurs threads ou la fonctionnalité intégrée de minuterie/gestionnaire d'interruption.

Si le lien du socket sortant est lent, des messages peuvent s'accumuler dans cette file d'attente. Vous pouvez utiliser connection::get\_buffered\_amount pour interroger la taille actuelle de la file d'attente des messages écrits afin de décider si vous souhaitez modifier votre comportement d'envoi.

**Ajoutez la méthode d'envoi à <tt>websocket\_endpoint</tt>**

Comme la méthode close, l'envoi commencera par rechercher l'ID de connexion donné dans la liste de connexions. Ensuite, une demande d'envoi est envoyée au handle de la connexion avec le message WebSocket spécifié et l'opcode texte. Enfin, nous enregistrons le message envoyé avec notre objet de métadonnées de connexion afin que notre commande show connection puisse imprimer plus tard une liste des messages envoyés.

void send(int id, std:: message sous forme de chaîne) {

websocketpp::lib::error\_code ec;

con\_list::iterator metadata\_it = m\_connection\_list.find(id);

if (metadata\_it == m\_connection\_list.end()) {

std::cout << "> Aucune connexion trouvée avec l'identifiant " << identifiant << std::endl;

retour;

}

m\_endpoint.send(metadata\_it->second->get\_hdl(), message, websocketpp::frame::opcode::text, ec);

si (ec) {

std::cout << "> Erreur lors de l'envoi du message : " << ec.message() << std::endl;

retour;

}

metadata\_it->second->record\_sent\_message(message);

}

**Ajouter une option d'envoi à la boucle de commande et au message d'aide**

Une option d'envoi est ajoutée à la boucle de commande. Il faut un identifiant de connexion et un message texte pour envoyer. Une entrée est également ajoutée au système d'aide pour décrire comment la nouvelle commande peut être utilisée.

else if (input.substr(0,4) == "envoyer"< /span>) {

std :: stringstream ss (entrée);

std :: chaîne cmd ;

int identifiant ;

std::string message = "";

ss>> cmd>> identifiant;

std::getline(ss,message);

endpoint.send(id, message);

}

**Ajoutez de la colle à <tt>connection\_metadata</tt> pour stocker les messages envoyés**

Afin de stocker les messages envoyés sur cette connexion, du code est ajouté à **[connection\_metadata](https://docs.websocketpp.org/classconnection__metadata.html)**. Cela inclut un nouveau membre de données std::vector<std::string> m\_messages pour garder une trace de tous les messages envoyés et reçus ainsi qu'une méthode pour ajouter un message envoyé dans cette liste :

void record\_sent\_message(std::string message) {

m\_messages.push\_back(">> " + message);

}

Enfin, l'opérateur de sortie des métadonnées de connexion est mis à jour pour imprimer également une liste des messages traités :

sortie << "> Messages traités : (" << data.m\_messages.size() << ") \n"< un i=4>;

std::vector<std::string>::const\_iterator it;

pour (it = data.m\_messages.begin(); it != data.m\_messages.end(); ++it) {

sortie << \*il ≪≪ "\n";

}

**Réception de messages**

Les messages sont reçus en enregistrant un gestionnaire de messages. Ce gestionnaire sera appelé une fois par message reçu et sa signature est void on\_message(websocketpp::connection\_hdl hdl, endpoint::message\_ptr msg). Le connection\_hdl, comme le paramètre similaire des autres gestionnaires, est un handle pour la connexion sur laquelle le message a été reçu. message\_ptr est un pointeur vers un objet qui peut être interrogé pour la charge utile du message, l'opcode et d'autres métadonnées. Notez que le type message\_ptr, ainsi que son type de message sous-jacent, dépendent de la façon dont votre point de terminaison est configuré et peuvent être différents pour différentes configurations.

**Ajoutez un gestionnaire de messages à la méthode <tt>connection\_metadata</tt>**

Le comportement de réception de messages que nous implémentons consistera à collecter tous les messages envoyés et reçus et à les imprimer dans l'ordre lorsque la commande show connection est exécutée. Les messages envoyés sont déjà ajoutés à cette liste. Nous ajoutons maintenant un gestionnaire de messages qui envoie également les messages reçus à la liste. Les messages texte sont envoyés tels quels. Les messages binaires sont d'abord convertis au format hexadécimal imprimable.

void on\_message([websocketpp::connection\_hdl](https://docs.websocketpp.org/namespacewebsocketpp.html" \l "a6b3d26a10ee7229b84b776786332631d) hdl, [client::message\_ptr](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a585ecbbfd9689d4e4229e4c8378bd672) msg) {

if (msg->get\_opcode() == websocketpp::frame::opcode::text) {

m\_messages.push\_back(msg->get\_payload());

} autre {

m\_messages.push\_back([websocketpp::utility::to\_hex](https://docs.websocketpp.org/namespacewebsocketpp_1_1utility.html#a26e5a26395d95d2f6bf3a9edb8d06dd2)(msg->get\_payload()));

}

}

Afin que ce gestionnaire soit appelé lorsque de nouveaux messages sont reçus, nous l'enregistrons également auprès de notre connexion. Notez que contrairement à la plupart des autres gestionnaires, le gestionnaire de messages possède deux paramètres et nécessite donc deux espaces réservés.

con->set\_message\_handler(websocketpp::lib::bind(

&connection\_metadata::on\_message,

métadonnées\_ptr,

websocketpp::lib::placeholders::\_1,

websocketpp::lib::placeholders::\_2

));

**Construire**

Il n'y a aucun changement dans les instructions de construction depuis l'étape 5

**Courir**

Dans cet exemple, nous nous connectons à l'exemple WebSocket++ echo\_server. Ce serveur répétera tout message que nous lui renverrons. Vous pouvez également essayer de tester cela avec le serveur echo à l'adresse ws://echo.websocket.org avec des résultats similaires.

Entrez la commande : connectez ws://localhost:9002

> Connexion créée avec l'identifiant 0

Entrez la commande : envoyer 0 exemple de message

Entrez la commande : affichez 0

> URI : ws://localhost:9002

> Statut : Ouvert

> Serveur distant : WebSocket++/0.4.0

> Erreur/raison de fermeture : N/A

> Messages traités : (2)

>> exemple de message

≪≪ exemple de message

**Étape 7**

*Utilisation de TLS/WebSockets sécurisés*

* Changer les inclusions
* lien vers les nouvelles dépendances de la bibliothèque
* Changer la configuration
* ajouter letls\_init\_handler
* configurer le contexte SSL pour le niveau de sécurité souhaité
* mélanger des connexions sécurisées et non sécurisées dans une seule application.

**Chapitre 2 : Fonctionnalités intermédiaires**

**Étape 8**

*Fonctionnalités de niveau intermédiaire*

* Négociation de sous-protocole
* Définition et lecture des en-têtes personnalisés
* Ping-pong
* Des procurations ?
* Définition de l'agent utilisateur
* Définition de l'origine
* Minuteries et sécurité
* Comportement de fermeture
* Envoyer un message à toutes les connexions

**Divers éléments ne savent pas s'ils doivent être inclus ici ou ailleurs ?**

flux de contrôle principal de Websocket++. Une poignée de main, suivie d'une scission en 2 volets de contrôle indépendants

* Prise de contact – utilise les informations spécifiées avant l'appel à endpoint::connect pour construire une demande de prise de contact WebSocket. – Transmettez la demande de prise de contact WebSocket à la stratégie de transport. La politique de transport détermine comment acheminer ces octets vers le point final jouant le rôle de serveur. Selon la politique de transport utilisée par votre point de terminaison, cette méthode sera différente. – Recevoir une réponse d'établissement de liaison du transport sous-jacent. Ceci est analysé et vérifié pour sa conformité à la RFC6455. Si la validation échoue, le gestionnaire d'échec est appelé. Sinon, le gestionnaire d'ouverture est appelé.
* À ce stade, le contrôle se divise en deux volets distincts. L'un qui lit les nouveaux octets de la politique de transport sur le canal entrant, l'autre qui accepte les nouveaux messages de l'application locale pour le tramage et l'écriture sur le canal de transport sortant.
* Lire le brin – Lire et traiter les nouveaux octets du transport – Si les octets contiennent au moins un message complet, répartissez chaque message en appelant le gestionnaire approprié. Il s'agit soit du gestionnaire de messages pour les messages de données, soit des gestionnaires ping/pong/close pour chaque message de contrôle respectif. Si aucun gestionnaire n'est enregistré pour un message particulier, celui-ci est ignoré. – Demander à la couche transport plus d’octets
* Brin d'écriture – Attendre les messages de l'application – Effectuer une vérification des erreurs sur l'entrée du message, – Tramer le message selon RFC6455 – Mettre le message en file d'attente pour l'envoi – Transmettre tous les messages en attente à la politique de transport pour la sortie – Lorsqu'il n'y a plus de messages à envoyer, revenir à en attendant

**Didacticiel d'exemple d'application de SERVEUR utilitaire**

# Introduction

Ce didacticiel fournit une discussion étape par étape sur la création d'un serveur WebSocket++ de base. Le produit final de ce didacticiel est l'exemple d'application **[utility\_server](https://docs.websocketpp.org/classutility__server.html)** de la section exemple. Ce serveur présente les fonctionnalités suivantes :

* Utiliser Asio Transport pour la mise en réseau
* Accepter plusieurs connexions WebSocket à la fois
* Lisez les messages entrants et effectuez quelques actions de base (écho, diffusion, télémétrie, commandes serveur) en fonction du chemin
* Utilisez le gestionnaire de validation pour rejeter les connexions à des chemins non valides
* Servir des réponses HTTP de base avec le gestionnaire http
* Quittez gracieusement le serveur
* Chiffrer les connexions avec TLS

Ce didacticiel est à jour à partir de la version 0.6.x de la bibliothèque.

# Chapitre 1 : Configuration initiale et configuration initiale Les bases

## Étape 1

Ajoutez des inclusions WebSocket++ et configurez un type de point de terminaison de serveur.

WebSocket++ comprend deux types d'objets principaux. Le point final et la connexion. Le point de terminaison crée et lance de nouvelles connexions et conserve les paramètres par défaut pour ces connexions. Les points de terminaison gèrent également toutes les ressources réseau partagées.

La connexion stocke les informations spécifiques à chaque session WebSocket.

**Remarque :** Une fois qu'une connexion est lancée, il n'y a aucun lien entre le point de terminaison et la connexion. Tous les paramètres par défaut sont copiés dans la nouvelle connexion par le point de terminaison. La modification des paramètres par défaut sur un point de terminaison n’affectera que les connexions futures.

Les connexions ne maintiennent pas de lien vers leur point de terminaison associé. Les points de terminaison ne conservent pas de liste des connexions en attente. Si votre application doit parcourir toutes les connexions, elle devra en conserver elle-même la liste.

Les points de terminaison WebSocket++ sont créés en combinant un rôle de point de terminaison avec une configuration de point de terminaison. Il existe deux types différents de rôles de point de terminaison, un pour les rôles client et un pour serveur dans une session WebSocket. Il s'agit d'un didacticiel de serveur, nous allons donc utiliser le rôle de serveur **[websocketpp::server](https://docs.websocketpp.org/classwebsocketpp_1_1server.html" \o "Server endpoint role based on the given config.)** qui est fourni par l'en-tête .<**[websocketpp/server.hpp](https://docs.websocketpp.org/server_8hpp_source.html)**>

### Terminologie : configuration du point de terminaison

Les points de terminaison WebSocket++ disposent d'un groupe de paramètres qui peuvent être configurés au moment de la compilation via le paramètre de modèle config. Une configuration est une structure qui contient des types et des constantes statiques utilisées pour produire un point de terminaison avec des propriétés spécifiques. En fonction de la configuration utilisée, le point de terminaison aura différentes méthodes disponibles et pourra avoir des dépendances tierces supplémentaires.

Le rôle de point de terminaison prend un paramètre de modèle appelé config qui est utilisé pour configurer le comportement du point de terminaison au moment de la compilation. Pour cet exemple, nous allons utiliser une configuration par défaut fournie par la bibliothèque appelée asio, fournie par . Il s'agit d'une configuration de serveur qui utilise la bibliothèque Asio pour assurer le transport réseau et ne prend pas en charge la sécurité basée sur TLS. Plus tard, nous verrons comment introduire la sécurité basée sur TLS dans une application WebSocket++, plus sur les autres configurations de stock et comment créer vos propres configurations personnalisées.<**[websocketpp/config/asio\_no\_tls.hpp](https://docs.websocketpp.org/asio__no__tls_8hpp_source.html)**>

Combinez une configuration avec un rôle de point de terminaison pour produire un point de terminaison entièrement configuré. Ce type sera fréquemment utilisé, je recommanderais donc un typedef ici.

typedef **[websocketpp::server](https://docs.websocketpp.org/classwebsocketpp_1_1server.html" \o "Server endpoint role based on the given config.)**<**[websocketpp::config::asio](https://docs.websocketpp.org/structwebsocketpp_1_1config_1_1asio.html" \o "Server config with asio transport and TLS disabled.)**> server

### <tt>utility\_server</tt> constructeur

Ce type de point de terminaison sera la base de l'objet **[utility\_server](https://docs.websocketpp.org/classutility__server.html)** qui gardera une trace de l'état du serveur. Au sein du constructeur **[utility\_server](https://docs.websocketpp.org/classutility__server.html)** plusieurs choses se produisent :

Tout d'abord, nous ajustons le comportement de journalisation des points de terminaison pour inclure tous les canaux de journalisation des erreurs et tous les canaux de journalisation des accès, à l'exception de la charge utile des trames, qui est particulièrement bruyante et généralement utile uniquement pour le débogage. [TODO : lien vers plus d'informations sur la journalisation]

m\_endpoint.set\_error\_channels([websocketpp::log::elevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1elevel.html" \l "a9b31ff708c221d314f9f4eb3ff2b1ad7));

m\_endpoint.set\_access\_channels([websocketpp::log::alevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html" \l "a853aa0b8976e53f3181af3bc398d493e) ^ [websocketpp:: log::alevel::frame\_payload](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html#aa38cfdf7a82f33cac319438462707e90));

Ensuite, nous initialisons le système de transport sous-jacent au point final. Cette méthode est spécifique au transport Asio et non au noyau WebSocket++. Il ne sera pas nécessaire ni présent dans les points de terminaison qui utilisent une configuration non-asio.

**Remarque :** Cet exemple utilise un Asio interne io\_service qui est géré par le point de terminaison lui-même. Il s'agit d'un arrangement simple adapté aux programmes dans lesquels WebSocket++ est le seul code utilisant Asio. Si vous avez un programme existant qui gère déjà un objet io\_service ou si vous souhaitez créer un nouveau programme dans lequel les gestionnaires WebSocket++ partagent un io\_service avec d'autres gestionnaires, vous pouvez transmettre le io\_service vous souhaite que WebSocket++ enregistre ses gestionnaires dans la méthode init\_asio() et il l'utilisera au lieu de générer et de gérer les siens. [À FAIRE : Lien FAQ à la place ?]

m\_endpoint.init\_asio();

### <tt>utility\_server::run</tt> méthode

En plus du constructeur, nous ajoutons également une méthode run qui configure le socket d'écoute, commence à accepter les connexions et démarre la boucle d'événements Asio io\_service.

// Écoute sur le port 9002

m\_endpoint.écouter(9002);

// Met en file d'attente une opération d'acceptation de connexion

m\_endpoint.start\_accept();

// Démarre la boucle d'exécution Asio io\_service

m\_endpoint.run();

La dernière ligne, m\_endpoint.run();, sera bloquée jusqu'à ce que le point de terminaison reçoive l'instruction d'arrêter d'écouter les nouvelles connexions. Pendant son exécution, il écoutera et traitera les nouvelles connexions, acceptera et traitera les nouvelles données et contrôlera les messages pour les connexions existantes. WebSocket++ utilise Asio dans un mode asynchrone où plusieurs connexions peuvent être gérées simultanément et efficacement au sein d'un seul thread.

### Construire

L'ajout de WebSocket++ a ajouté quelques dépendances à notre programme qui doivent être traitées dans le système de build. Tout d’abord, les en-têtes de bibliothèque WebSocket++ doivent se trouver dans le chemin de recherche d’inclusion de votre système de build. La manière exacte dont cela est effectué dépend de l'endroit où vous avez installé les en-têtes WebSocket++ et du système de build que vous utilisez.

Pour le reste de ce didacticiel, nous allons supposer un environnement de construction C++11. WebSocket++ fonctionnera avec les systèmes antérieurs à C++11 si votre système de build a accès à une version récente des en-têtes de la bibliothèque Boost.

Enfin, pour utiliser la configuration de transport Asio, nous devons importer la bibliothèque Asio. Il y a deux options ici. Si vous avez accès à un environnement de build C++11, la version autonome de [http://think-async.com](http://think-async.com/) est une bonne option. Cette bibliothèque d'en-tête uniquement n'apporte aucune dépendance particulière et garantit que vous disposez de la dernière version d'Asio. Si vous ne disposez pas d'un environnement de build C++11 ou si vous avez déjà intégré les bibliothèques Boost, vous pouvez également utiliser la version d'Asio fournie avec Boost.

Pour utiliser Asio autonome, assurez-vous que les en-têtes Asio se trouvent dans votre chemin d'inclusion et définissez ASIO\_STANDALONE. Pour utiliser Boost Asio, assurez-vous que les en-têtes Boost se trouvent dans votre chemin d'inclusion et que vous établissez un lien vers la bibliothèque boost\_system.

c++ -std=c++11 step1.cpp (Asio autonome) OU c++ -std=c++11 step1.cpp -lboost\_system (Boost Asio)

### Code jusqu'à présent

// La définition ASIO\_STANDALONE est nécessaire pour utiliser la version autonome d'Asio.

// Supprimez si vous utilisez Boost Asio.

#définir ASIO\_STANDALONE

#include <websocketpp/config/asio\_no\_tls.hpp>

#include <websocketpp/server.hpp>

#include <fonctionnel>

typedef [websocketpp::server<websocketpp::config::asio>](https://docs.websocketpp.org/classwebsocketpp_1_1server.html) [serveur](https://docs.websocketpp.org/classwebsocketpp_1_1server.html);

classe [utility\_server](https://docs.websocketpp.org/classutility__server.html) {

public :

[utility\_server](https://docs.websocketpp.org/classutility__server.html)() {

// Définir les paramètres de journalisation

m\_endpoint.[set\_error\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a8292bcdca9344b57af1b0964ff7fc778)([websocketpp::log::elevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1elevel.html" \l "a9b31ff708c221d314f9f4eb3ff2b1ad7) );

m\_endpoint.[set\_access\_channels](https://docs.websocketpp.org/classwebsocketpp_1_1endpoint.html#a5d7da609ebd2f797e5e67b6d050ebc59)([websocketpp::log::alevel::all](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html" \l "a853aa0b8976e53f3181af3bc398d493e) ^ [websocketpp::log::alevel::frame\_payload](https://docs.websocketpp.org/structwebsocketpp_1_1log_1_1alevel.html#aa38cfdf7a82f33cac319438462707e90));

// Initialise Asio

m\_endpoint.init\_asio();

}

void run() {

// Écoute sur le port 9002

m\_endpoint.écouter(9002);

// Met en file d'attente une opération d'acceptation de connexion

m\_endpoint.[start\_accept](https://docs.websocketpp.org/classwebsocketpp_1_1server.html#a0204a7d444144f7ea5b8bbcf14689fc1)();

// Démarre la boucle d'exécution Asio io\_service

m\_endpoint.run();

}

privé :

[serveur](https://docs.websocketpp.org/classwebsocketpp_1_1server.html) m\_endpoint;

} ;

int main() {

[utility\_server](https://docs.websocketpp.org/classutility__server.html) s;

s.run();

retour 0;

}

## Étape 2

Configurer un gestionnaire de messages pour renvoyer toutes les réponses à l'utilisateur d'origine

### Définition d'un gestionnaire de messages

##### Terminologie : enregistrement des gestionnaires

WebSocket++ fournit un certain nombre de points d'exécution où vous pouvez vous inscrire pour exécuter un gestionnaire. Les points disponibles pour votre point de terminaison dépendront de sa configuration. Les gestionnaires TLS n'existeront pas sur les points de terminaison non TLS par exemple. Une liste complète des gestionnaires est disponible sur <http://www.zaphoyd.com/websocketpp/manual/reference/handler-list>.

Les gestionnaires peuvent être enregistrés au niveau du point de terminaison et au niveau de la connexion. Les gestionnaires de points de terminaison sont copiés dans les nouvelles connexions au fur et à mesure de leur création. La modification d’un gestionnaire de point de terminaison affectera uniquement les connexions futures. Les gestionnaires enregistrés au niveau de la connexion seront liés uniquement à cette connexion spécifique.

La signature des méthodes de liaison du gestionnaire est la même pour les points de terminaison et les connexions. Le format est : set\_\*\_handler(...). Où \* est le nom du gestionnaire. Par exemple, set\_open\_handler(...) définira le gestionnaire à appeler lorsqu'une nouvelle connexion est ouverte. set\_fail\_handler(...) définira le gestionnaire à appeler lorsqu'une connexion échoue.

Tous les gestionnaires prennent un argument, un type appelable qui peut être converti en std::function avec le nombre et le type d'arguments corrects. Vous pouvez transmettre des fonctions gratuites, des foncteurs et des Lambdas avec des listes d'arguments correspondantes en tant que gestionnaires. De plus, vous pouvez utiliser std::bind (ou boost::bind) pour enregistrer des fonctions avec des listes d'arguments qui ne correspondent pas. Ceci est utile pour transmettre des paramètres supplémentaires non présents dans la signature du gestionnaire ou dans les fonctions membres qui doivent porter un « ce » ; pointeur.

La signature de fonction de chaque gestionnaire peut être recherchée dans la liste ci-dessus dans le manuel. En général, tous les gestionnaires incluent le connection\_hdl identifiant la connexion à laquelle il est associé comme premier paramètre. Certains gestionnaires (tels que le gestionnaire de messages) incluent des paramètres supplémentaires. La plupart des gestionnaires ont une valeur de retour vide, mais certains (validate, ping, tls\_init) n'en ont pas. Les significations spécifiques des valeurs de retour sont documentées dans la liste des gestionnaires liée ci-dessus.

## Étape 3

la gestion des erreurs

## Étape 4

Configurer des gestionnaires d'ouverture et de fermeture et une structure de données de connexion

## Étape 5

Modifiez le gestionnaire de messages pour les connexions basées sur l'URI et ajoutez un gestionnaire de validation pour rejeter les URI non valides.

## Étape 6

Ajoutez quelques commandes d'administration (rapportez le nombre total de clients, arrêtez proprement le serveur)

## Étape 7

Ajouter quelques commandes de diffusion

## Étape 8

Ajouter TLS