Client

Jump to bottom

Gil Maimon edited this page on Jan 16, 2021 · 12 revisions

Go back to API Reference

A Websocket is a 2-way communication channel between 2 endpoints. In order to initiate a connection, `Client` handpoints connect to `Server` endpoints.

`WebsocketsClient` lets you connect to servers listening for websockets connections and communicate with them.

WebsocketsClient

To include `WebsocketsClient`:

```
#include <tiny_websockets/client.hpp>
```

`WebsocketsClient` constructor takes an abstract `network::TcpClient` but it is created by default according to the platform you compile on.

So, in order to create a `WebsocketsClient` instance just use the deafult constructor:

```
websockets::WebsocketsClient client;

// use client
// ...
```

Table Of Contents

Managing The Connection

You can check if the client is connected (and whether the connection is still open) using:

```
bool available();
```

If `client.available()` returns *true*, the client is connected and ready to send and recv messages.

In order to close the connection, call `close`.

```
void close(CloseReason reason = CloseReason_NormalClosure);
```

The `reason` parameter is optional. It indicates the reason for closing the connection. Read here more about Close Reason Codes.

After closing the connection, sending and trying to receive messages will always fail.

Connecting to a Server

To connect to a server, use:

```
bool connect(std::string url);
bool connect(std::string host, int port, std::string path);
```

Return Value

A boolean value that indicates if the connection was successfull.

true: connection was successfull
false: connection failed

Example

```
// First Form
client.connect("http://server.mydomain.org:8080/echo");
```

```
client.connect("ws://server.mydomain.org:8080/echo");
client.connect("server.mydomain.org");

// Second Form
client.connect("server.mydomain.org", 8080, "/echo");
```

Sending Messages

To send data and binary messages you can use:

```
// Text Messages
bool send(std::string data);
bool send(const char* data, size_t len);

// Binary Messages
bool sendBinary(std::string data);
bool sendBinary(const char* data, size_t len);
```

Return Value

A boolean value that indicates if message was successfully sent.

true: successfully sent
false: failed to send

Example

```
client.send("Hello Server");
client.sendBinary(buff, 64);
```

Sending Pings and Pongs

To send data and binary messages you can use:

```
bool ping(std::string data = "");
bool pong(std::string data = "");
```

Return Value

A boolean value that indicates if the ping/pong was successfully sent.

true: successfully sent
false: failed to send

Example

```
client.ping();
client.pong();
client.ping("Ping Data");
client.pong("Pong Data");
```

Receiving Messages and Events - Callbacks

Message can be received either using a blocking or non-blocking interface.

```
// Messages Callback
void onMessage(MessageCallback callback);

// Acceptable signatures:
typedef std::function<void(WebsocketsClient&, WebsocketsMessage)> MessageCallback; // Comp
typedef std::function<void(WebsocketsMessage)> MessageCallback; // Short Form

// Events Listener
void onEvent(EventCallback callback);

// Acceptable signatures:
typedef std::function<void(WebsocketsClient&, WebsocketsEvent, std::string)> EventCallback
typedef std::function<void(WebsocketsEvent, std::string)> EventCallback; // Short Form
```

The **onMessage** callback will be called every time a message will be received. In order for the client to proccess new messages the users must call `poll()`.

The **onEvent** callback will be called every time a control frame will be received or when an event happens internally (for example the user closing the socket). In order for the client to proccess new frames the users must call `poll()`.

Example

```
// setup messages callback
client.onMessage([](WebsocketsClient& client, WebsocketsMessage msg) {
  if(msg.isText()) {
    std::cout << "Got Text Message: " << msg.data() << std::endl;
  }
});

// setup events callback
client.onEvent([](WebsocketsClient& client, WebsocketsEvent event, std::string payload) {
  switch(event) {
    case WebsocketsEvent::ConnectionOpened:
```

```
        // Dispatched when connecting to a server
        break;
      case WebsocketsEvent::GotPing:
        // Dispatched when a ping frame arrives
        break;
      case WebsocketsEvent::GotPong:
        // Dispatched when a pong frame arrives
        break;
      case WebsocketsEvent::ConnectionClosed:
        // Dispatched when the connection is closed (either
        // by the user or after some error or event)
        break;
    }
  });

  // check for changes
  while(true) {
    client.poll();
  }
```

Receiving Messages and Events - Blocking

The blocking interface removes the need for calling `poll()`:

```
WebsocketsMessage readBlocking();
```

`readBlocking` will return the first message **or event** received. `readBlocking` can also return `Ping`, `Pong` and `Close` messages.

Note: in case of socket errors, readBlocking will return with no actual messages. You will need to test that `msg.isEmpty()` is *false* to make sure the message returned is not a default (error) value.

Example

```
// wait for a message and print it
auto msg = client.readBlocking();
if(msg.isText()) {
  std::cout << "Got Text Message: " << msg.data() << std::endl;
} else if(msg.isPing()) {
  std::cout << "Got Ping with payload: " << msg.data() << std::endl;
}
```

Streaming - Sending Fragmented Messages

`WebsocketsClient` provides a simple interface for streaming messages (sending a fragmented message). For streaming, use the following methods:

```
// to start streaming text/binary
bool stream(std::string data = "");
bool streamBinary(std::string data = "");

// to end the stream
bool end(std::string data = "");
```

For streaming:

1. call `stream` or `streamBinary` (you don't have to add a payload).
2. call `send` or `sendBinary` for every chunk you want to send.
3. call `end` once you are done streaming the message.

Every message will be sent independently as a continuation frame.

Example

```
client.stream();
client.send("This");
client.send("Message");
client.send("Will Be");
client.send("Fragmented!");
client.end();
```

Handling Fragmented Messages

By default, when a `WebsocketsClient` gets fragmented messages it waits for all the parts before dispatching any callback or returning from `readBlocking`.

If you want to, you can choose to be updated for each continuation frame that arrives to the client. You can do that by setting the `FragmentsPolicy`.

```
enum FragmentsPolicy {
  FragmentsPolicy_Aggregate,
  FragmentsPolicy_Notify
};

void setFragmentsPolicy(FragmentsPolicy newPolicy);
FragmentsPolicy getFragmentsPolicy();
```

- `FragmentsPolicy_Aggregate` - is the default policy, it will wait for complete messages before dispatching callbacks and returning from `readBlocking`. That way, the messages the user handles are always complete.

- `FragmentsPolicy_Notify` - setting the policy to `Notify` will make the client notify the user (via callbacks and `readBlocking`) for every fragment that arrives. **It is the user's responsibility to collect the messages and handle them as a complete message**.

Close Reason Codes

The `reason` parameter is optional. It indicates the reason for closing the connection. Closure reasons are described in the RFC. By default the close reason is `Normal Closure`.

When a `WebsocketsClient` is destructed without a call to `close()` the close reason will be `CloseReason_GoingAway`.

If a client is already closed (either by you or by the server), you can check the close reason by calling `client.getCloseReason()`.

The possible close reasons are:

```
enum CloseReason {
  CloseReason_None                =       -1,
  CloseReason_NormalClosure       =       1000,
  CloseReason_GoingAway           =       1001,
  CloseReason_ProtocolError       =       1002,
  CloseReason_UnsupportedData     =       1003,
  CloseReason_NoStatusRcvd        =       1005,
  CloseReason_AbnormalClosure     =       1006,
  CloseReason_InvalidPayloadData  =       1007,
  CloseReason_PolicyViolation     =       1008,
  CloseReason_MessageTooBig       =       1009,
  CloseReason_InternalServerError =       1011,
};
```

Example

For example, lets say in our application we only wants to accept text messages (no binary data). The following code can be used:

```
client.onMessage([&](WebsocketsMessage message) {
  if(message.isBinary()) {
    client.close(CloseReason_UnsupportedData);
    return;
  }
```

```
  // Handle message
});

// Handle client and call client.poll()
```

Examples

Interactive demo using websocket.org

```
/*
Interactive and basic Websockets Client that connect to a public echo server

After running this demo, there will be a websockets client connected
to `echo.websocket.org` and every message the user will enter will be
sent to the server. Incoming messages will be printed once the user
enters an input (or an empty line, just an Enter)
Enter "exit" to close the connection and end the program.

The code:
  1. Sets up a client connection
  2. Reads an input from the user
    2-1. If the user didnt enter an empty line, the client sends the message to the server
    2-2. If the user enters "exit", the program closes the connection
  3. Polls for incoming messages and events.
*/

#include <tiny_websockets/client.hpp>
#include <tiny_websockets/server.hpp>
#include <iostream>

using namespace websockets;

int main() {
  WebsocketsClient client;
  client.connect("ws://echo.websocket.org/");

  client.onMessage([&](WebsocketsClient&, WebsocketsMessage message){
    std::cout << "Got Data: " << message.data() << std::endl;
  });

  WSString line;
  while(client.available()) {
    std::cout << "Enter input: ";
    std::getline(std::cin, line);

    if(line != "") {
      if(line == "exit") client.close();
      else {
        client.poll();
        client.send(line);
```

```
        }
      }
      client.poll();
    }
    std::cout << "Exited Gracefully" << std::endl;
}
```

Go back to API Reference

Written by Gil Maimon @ 2019

▸ **Pages** 8

Introduction

- Home
- What Are Websockets
- Getting Started
- Examples

API Reference

- Message
- Client
- Server

Clone this wiki locally

```
https://github.com/gilmaimon/TinyWebsockets.wiki.git
```