# socketio/socket.io-client-cpp

github.com/algo.html

## API

### *Overview*

There're just 3 roles in this library - `socket` , `client` and `message` .

`client` is for physical connection while `socket` is for "namespace" (which is like a logical channel), which means one `socket` paired with one namespace, and one `client` paired with one physical connection.

Since a physical connection can have multiple namespaces (which is called multiplex), a `client` object may have multiple `socket` objects, each of which is bound to a distinct `namespace` .

Use `client` to setup the connection to the server, manange the connection status, also session id for the connection.

Use `socket` to send messages under namespace and receives messages in the namespace, also handle special types of message.

The `message` is just about the content you want to send, with text, binary or structured combinations.

### *Socket*

#### Constructors

Sockets are all managed by `client` , no public constructors.

You can get it's pointer by `client.socket(namespace)` .

#### Event Emitter

`void emit(std::string const& name, message::list const& msglist, std::function<void (message::ptr const&)> const& ack)`

Universal event emition interface, by applying implicit conversion magic, it is backward compatible with all previous `emit` interfaces.

#### Event Bindings

`void on(std::string const& event_name,event_listener const& func)`

`void on(std::string const& event_name,event_listener_aux const& func)`

Bind a callback to specified event name. Same as `socket.on()` function in JS, `event_listener` is for full content event object, `event_listener_aux` is for convenience.

`void off(std::string const& event_name)`

Unbind the event callback with specified name.

`void off_all()`

Clear all event bindings (not including the error listener).

`void on_error(error_listener const& l)`

Bind the error handler for socket.io error messages.

`void off_error()`

Unbind the error handler.

```
//event object:
class event
{
public:
    const std::string& get_nsp() const;

    const std::string& get_name() const;

    const message::ptr& get_message() const;

    bool need_ack() const;

    void put_ack_message(message::ptr const& ack_message);

    message::ptr const& get_ack_message() const;
   ...
};
//event listener declare:
typedef std::function<void(const std::string& name,message::ptr const& message,bool need_ack, message::ptr& ack_message)> event_listener_aux;

typedef std::function<void(event& event)> event_listener;

typedef std::function<void(message::ptr const& message)> error_listener;
```

## Connect and close socket

`connect` will happen for existing `socket`s automatically when `client` have opened up the physical connection.

`socket` opened with connected `client` will connect to its namespace immediately.

`void close()`

Positively disconnect from namespace.

## Get name of namespace

`std::string const& get_namespace() const`

Get current namespace name which the client is inside.

## *Client*

### Constructors

`client()` default constructor.

### Connection Listeners

`void set_open_listener(con_listener const& l)`

Call when websocket is open, especially means good connectivity.

`void set_fail_listener(con_listener const& l)`

Call when failed in connecting.

`void set_close_listener(close_listener const& l)`

Call when closed or drop. See `client::close_reason`

```
//connection listener declare:
enum close_reason
{
    close_reason_normal,
    close_reason_drop
};
typedef std::function<void(void)> con_listener;

typedef std::function<void(close_reason const& reason)> close_listener;
```

### Socket listeners

`void set_socket_open_listener(socket_listener const& l)`

Set listener for socket connect event, called when any sockets being ready to send message.

`void set_socket_close_listener(socket_listener const& l)`

Set listener for socket close event, called when any sockets being closed, afterward, corresponding `socket` object will be cleared from client.

```
    //socket_listener declare:
    typedef std::function<void(std::string const& nsp)> socket_listener;
```

### Connect and Close

`void connect(const std::string& uri)`

Connect to socket.io server, e.g., `client.connect("ws://localhost:3000");`

`void close()`

Close the client, return immediately.

`void sync_close()`

Close the client, return until it is really closed.

`bool opened() const`

Check if client's connection is opened.

### Transparent reconnecting

`void set_reconnect_attempts(int attempts)`

Set max reconnect attempts, set to 0 to disable transparent reconnecting.

`void set_reconnect_delay(unsigned millis)`

Set minimum delay for reconnecting, this is the delay for 1st reconnecting attempt, then the delay duration grows by attempts made.

`void set_reconnect_delay_max(unsigned millis)`

Set maximum delay for reconnecting.

`void set_reconnecting_listener(con_listener const& l)`

Set listener for reconnecting is in process.

`void set_reconnect_listener(reconnect_listener const& l)`

Set listener for reconnecting event, called once a delayed connecting is scheduled.

### Namespace

`socket::ptr socket(std::string const& nsp)`

Get a pointer to a socket which is paired with the specified namespace.

### Session ID

`std::string const& get_sessionid() const`

Get socket.io session id.

### *Message*

`message` Base class of all message object.

`int_message` message contains a 64-bit integer.

`double_message` message contains a double.

`string_message` message contains a string.

`array_message` message contains a `vector<message::ptr>`.

`object_message` message contains a `map<string,message::ptr>`.

`message::ptr` pointer to `message` object, it will be one of its derived classes, judge by `message.get_flag()`.

All designated constructor of `message` objects is hidden, you need to create message and get the `message::ptr` by `[derived]_message:create()`.