# WebSocket Interface Design Document

## 1. Overview

This document describes the implementation of a WebSocket interface for bidirectional communication between a C++ application and web browsers. The system enables real-time data exchange and remote control capabilities.

### 1.1 Key Components

- WebSocket Server/Client Interface (C++)

- Message Protocol (JSON-based)

- Event Handling System

- Connection Management

## 2. Technical Architecture

### 2.1 Class Structure

```

net::IWebSocket

├── WebSocketImpl (PIMPL implementation)

│ ├── Server Management

│ ├── Client Management

│ ├── Message Handling

│ └── Connection Management

└── Public Interface Methods

```

### 2.2 Dependencies

- WebSocket++ (websocketpp)

- JsonCpp (jsoncpp)

- C++11 or later

- Asio (standalone or boost)

## 3. Message Protocol

### 3.1 Message Structure

All messages follow this base structure:

```json

{

"type": "<message-type>",

"id": "<uuid>",

"timestamp": <unix-ms-timestamp>,

"payload": {

// Type-specific content

}

}

```

### 3.2 Message Types

#### 3.2.1 Command Messages (Browser → C++)

```json

{

"type": "command",

"id": "cmd-uuid",

"timestamp": 1699312435000,

"payload": {

"command": {

"name": "getData",

"parameters": {

"startTime": 1699312435000,

"endTime": 1699312435000,

"filter": "temperature"

}

}

}

}

```

#### 3.2.2 Event Messages (C++ → Browser)

```json

{

"type": "event",

"id": "evt-uuid",

"timestamp": 1699312435000,

"payload": {

"event": {

"name": "statusChanged",

"data": {

"status": "running",

"details": {

"cpuUsage": 45.2,

"memoryUsage": 1024

}

}

}

}

}

```

#### 3.2.3 Response Messages (Both directions)

```json

{

"type": "response",

"id": "rsp-uuid",

"timestamp": 1699312435000,

"payload": {

"response": {

"success": true,

"data": {

// Response-specific data

}

}

}

}

```

#### 3.2.4 Error Messages (Both directions)

```json

{

"type": "error",

"id": "err-uuid",

"timestamp": 1699312435000,

"payload": {

"error": {

"code": 1001,

"message": "Invalid parameters",

"details": {

"field": "startTime",

"reason": "Must be less than endTime"

}

}

}

}

```

## 4. Implementation Guide

### 4.1 Creating Messages (C++)

#### 4.1.1 Command Message

```cpp

Json::Value createCommand(const std::string& commandName,

const Json::Value& parameters) {

Json::Value message;

message["type"] = "command";

message["id"] = generateUUID();

message["timestamp"] = getCurrentTimestamp();

Json::Value& command = message["payload"]["command"];

command["name"] = commandName;

if (!parameters.isNull()) {

command["parameters"] = parameters;

}

return message;

}

```

#### 4.1.2 Event Message

```cpp

Json::Value createEvent(const std::string& eventName,

const Json::Value& data) {

Json::Value message;

message["type"] = "event";

message["id"] = generateUUID();

message["timestamp"] = getCurrentTimestamp();

Json::Value& event = message["payload"]["event"];

event["name"] = eventName;

if (!data.isNull()) {

event["data"] = data;

}

return message;

}

```

### 4.2 Parsing Messages (C++)

#### 4.2.1 Message Parser

```cpp

class MessageParser {

public:

static bool parse(const std::string& jsonStr, Json::Value& root) {

Json::Reader reader;

if (!reader.parse(jsonStr, root)) {

return false;

}

// Validate required fields

if (!validateMessage(root)) {

return false;

}

return true;

}

private:

static bool validateMessage(const Json::Value& message) {

if (!message.isMember("type") || !message.isMember("id") ||

!message.isMember("timestamp") || !message.isMember("payload")) {

return false;

}

return true;

}

};

```

### 4.3 Message Handling (C++)

#### 4.3.1 Message Handler Implementation

```cpp

class MessageHandler {

public:

void handleMessage(const std::string& jsonStr) {

Json::Value message;

if (!MessageParser::parse(jsonStr, message)) {

sendError("Invalid message format", 1001);

return;

}

std::string type = message["type"].asString();

if (type == "command") {

handleCommand(message);

} else if (type == "event") {

handleEvent(message);

} else if (type == "response") {

handleResponse(message);

} else if (type == "error") {

handleError(message);

}

}

private:

void handleCommand(const Json::Value& message) {

const Json::Value& command = message["payload"]["command"];

std::string cmdName = command["name"].asString();

// Command routing

if (cmdName == "getData") {

handleGetDataCommand(command);

} else if (cmdName == "setConfig") {

handleSetConfigCommand(command);

}

// ... other commands

}

// Implement other handlers...

};

```

## 5. Client-Side Implementation (JavaScript)

### 5.1 WebSocket Client Class

```javascript

class WebSocketClient {

constructor(url) {

this.ws = new WebSocket(url);

this.messageHandlers = new Map();

this.setupEventHandlers();

}

setupEventHandlers() {

this.ws.onmessage = (event) => {

const message = JSON.parse(event.data);

this.handleMessage(message);

};

this.ws.onopen = () => {

this.onConnected();

};

this.ws.onclose = () => {

this.onDisconnected();

};

}

sendCommand(commandName, parameters = {}) {

const message = {

type: 'command',

id: this.generateUUID(),

timestamp: Date.now(),

payload: {

command: {

name: commandName,

parameters: parameters

}

}

};

this.ws.send(JSON.stringify(message));

}

handleMessage(message) {

const handler = this.messageHandlers.get(message.type);

if (handler) {

handler(message);

}

}

}

```

## 6. Error Handling

### 6.1 Error Codes

```

1000-1999: Protocol Errors

- 1001: Invalid message format

- 1002: Missing required fields

- 1003: Invalid message type

2000-2999: Command Errors

- 2001: Unknown command

- 2002: Invalid parameters

- 2003: Command execution failed

3000-3999: Connection Errors

- 3001: Connection lost

- 3002: Reconnection failed

- 3003: Authentication failed

```

### 6.2 Error Handling Example

```cpp

void handleError(const std::string& message, int code) {

Json::Value errorMsg;

errorMsg["type"] = "error";

errorMsg["id"] = generateUUID();

errorMsg["timestamp"] = getCurrentTimestamp();

errorMsg["payload"]["error"]["code"] = code;

errorMsg["payload"]["error"]["message"] = message;

sendMessage(errorMsg);

}

```

## 7. Usage Examples

### 7.1 C++ Application

```cpp

int main() {

net::IWebSocket ws;

ws.setMessageCallback([](const Json::Value& message) {

// Handle incoming messages

if (message["type"].asString() == "command") {

// Process command

const Json::Value& cmd = message["payload"]["command"];

processCommand(cmd["name"].asString(), cmd["parameters"]);

}

});

ws.setStatusCallback([](const std::string& status, bool isError) {

std::cout << "Status: " << status <<

(isError ? " (Error)" : "") << std::endl;

});

if (ws.startServer(9002)) {

// Server started successfully

std::cout << "WebSocket server running on port 9002" << std::endl;

// Main application loop

while (true) {

// Your application logic here

// Send updates to clients when needed

Json::Value event = createEvent("statusUpdate", getStatusData());

ws.sendMessage(event);

std::this\_thread::sleep\_for(std::chrono::seconds(1));

}

}

return 0;

}

```

### 7.2 Browser Client

```javascript

const client = new WebSocketClient('ws://localhost:9002');

// Register message handlers

client.messageHandlers.set('event', (message) => {

const event = message.payload.event;

switch (event.name) {

case 'statusUpdate':

updateUI(event.data);

break;

case 'error':

showError(event.data);

break;

}

});

// Send command to C++ application

function requestData() {

client.sendCommand('getData', {

startTime: Date.now() - 3600000, // Last hour

endTime: Date.now()

});

}

```

## 8. Best Practices

### 8.1 Message Handling

- Always validate message format before processing

- Include error handling for all operations

- Use type-safe message creation helpers

- Implement message acknowledgment for critical operations

### 8.2 Connection Management

- Implement automatic reconnection

- Handle connection errors gracefully

- Monitor connection health with heartbeats

- Clean up resources properly on disconnect

### 8.3 Performance

- Use message queuing for high-frequency updates

- Implement message batching when appropriate

- Monitor message size and frequency

- Implement rate limiting if needed

## 9. Security Considerations

### 9.1 Recommendations

- Implement TLS for production environments

- Validate all input messages

- Implement authentication for sensitive operations

- Rate limit connections and messages

- Sanitize all data before processing

- Implement proper access controls