Техническая документация по Клиент-Серверному Взаимодействию

СОДЕРЖАНИЕ

[1 ВВЕДЕНИЕ 3](#_Toc184819363)

[2 АРХИТЕКТУРА 4](#_Toc184819364)

[3 КОМПОНЕНТЫ СЕРВЕРА 5](#_Toc184819365)

[3.1 Класс WebServer 5](#_Toc184819366)

[3.2 Класс clientAI 6](#_Toc184819367)

[3.3 Класс culcradar 7](#_Toc184819368)

[3.4 Класс radarCore 8](#_Toc184819369)

[3.5 Утилиты calctools 9](#_Toc184819370)

[4 ПРОТОКОЛ ВЗАИМОДЕЙСТВИЯ 11](#_Toc184819371)

[4.1 Формат Сообщений 11](#_Toc184819372)

[4.2 Этапы Взаимодействия 13](#_Toc184819373)

[4.3 Клиентская сторона взаимодействия 14](#_Toc184819374)

[5 УПРАВЛЕНИЕ ЗАДАЧАМИ И ПОТОКАМИ 16](#_Toc184819379)

[6 ОБРАБОТКА КОМАНД 18](#_Toc184819380)

[7 РАБОТА С ФАЙЛАМИ 22](#_Toc184819384)

[8 ОБРАБОТКА ОШИБОК 25](#_Toc184819388)

[9 ЛОГИРОВАНИЕ 27](#_Toc184819393)

# 1 ВВЕДЕНИЕ

Данная документация описывает клиент-серверное взаимодействие для приложения, разработанного с использованием фреймворка Qt на языке C++. Серверная часть реализует WebSocket-сервер, который обрабатывает подключения клиентов, выполняет вычислительные задачи, связанные с обработкой радиопортретов, и взаимодействует с клиентами через обмен JSON-сообщениями.

Цель документации — предоставить подробное описание архитектуры сервера, его компонентов, протокола взаимодействия с клиентами, а также процессов управления задачами и обработкой команд.

# 2 АРХИТЕКТУРА

Система представляет собой клиент-серверное приложение, где сервер реализован на базе Qt WebSockets (QWebSocketServer). Клиенты подключаются к серверу, проходят аутентификацию, отправляют данные для обработки, получают результаты вычислений и могут отправлять команды управления.

Основные компоненты сервера:

* WebSocket Server (WebServer): управляет подключениями клиентов и обработкой сообщений.
* Client Handler (clientAI): содержит информацию о каждом подключенном клиенте.
* Вычислительный Ядро (radarCore и culcradar): выполняет задачи расчета радиопортретов.
* Утилиты (calctools): обеспечивают логирование и вспомогательные функции.

# 3 КОМПОНЕНТЫ СЕРВЕРА

## 3.1 Класс WebServer

Расположение: webserver.cpp

Описание: Основной класс, реализующий WebSocket-сервер. Управляет подключениями клиентов, обработкой входящих сообщений, аутентификацией и управлением вычислительными задачами.

Основные Члены:

* QWebSocketServer \*m\_pWebSocketServer: Объект WebSocket-сервера.
* QList<QWebSocket\*> m\_clients: Список подключенных клиентов.
* QList<clientAI\*> m\_client\_list: Список объектов clientAI для каждого клиента.
* QList<radarCore\*> task\_list: Список активных вычислительных задач.
* Флаги управления сохранением данных (например, SAVE\_INPUT\_DATA\_TO\_FILES, SAVE\_MODEL и др.)

Основные Методы:

* Конструктор и Деструктор
  + WebServer(quint16 port, QObject \*parent): Инициализирует сервер на указанном порту, устанавливает соединения сигналов и слотов.
  + ~WebServer(): Закрывает сервер и освобождает ресурсы.
* Управление Подключениями
  + void onNewConnection(): Обрабатывает новые подключения клиентов.
  + void socketDisconnected(): Обрабатывает отключение клиентов.
* Обработка Сообщений
  + void processMessage(const QString &message): Обрабатывает входящие сообщения от клиентов, разбирает JSON и вызывает соответствующие методы.
* Аутентификация
  + void userVerification(QString login, QString password, QWebSocket \*pSender): Проверяет учетные данные клиента и устанавливает статус аутентификации.
* Управление Вычислительными Задачами
  + void loadRadarData(QJsonDocument &doc, QJsonObject &jsonObject, QWebSocket \*pSender): Принимает данные от клиента и инициирует вычислительную задачу.
  + void handlerCmd(QString command, QJsonArray \*params, QWebSocket \*pSender): Обрабатывает команды управления от клиента (например, пауза, продолжение, остановка).
  + bool task\_kill(QWebSocket \*pSender): Останавливает и удаляет вычислительную задачу для конкретного клиента.
* Вспомогательные Методы
  + QString GetRandomString(): Генерирует случайную строку для идентификации клиента.
  + void webServerAnswer(QString answer, QWebSocket \*pClient): Отправляет ответное сообщение клиенту в формате JSON.

Ключевые Связи:

* Соединяется с сигналами нового подключения (newConnection), отключения (disconnected) и приема сообщений (textMessageReceived) из QWebSocketServer.

## 3.2 Класс clientAI

Расположение: clientai.cpp

Описание: Класс, представляющий подключенного клиента. Содержит информацию об идентификаторе клиента, статусе аутентификации и соединении WebSocket.

Основные Члены:

* QString id: Уникальный идентификатор клиента.
* QString login: Логин клиента.
* bool authStatus: Статус аутентификации.
* QWebSocket \*pClient: Указатель на WebSocket-соединение клиента.
* QDate dateConnect: Дата подключения.
* QTime timeConnect: Время подключения.

Основные Методы:

* Конструктор
  + clientAI(QObject \*parent): Инициализирует объект.
* Аутентификация
  + void auth(): Проверяет статус аутентификации и закрывает соединение, если клиент не аутентифицирован.

Особенности:

* Использует механизм таймера для выполнения метода auth через 1 секунду после подключения.

## 3.3 Класс culcradar

Расположение: CulcRadar.cpp

Описание: Класс, отвечающий за вычисления радиопортрета. Обрабатывает входные данные, выполняет FFT и формирует результаты.

Основные Члены:

* Параметры Вычислений
  + double Lmax: Максимальная длина.
  + double stepX, stepY, stepZ: Шаги по осям.
  + bool boolX, boolY, boolZ: Флаги активации осей.
  + rVect Nin, Nout: Векторы направления.
  + QVector<QVector<QVector<cVect>>> vEout: Трехмерный массив результатов.
* Методы Вычислений
  + int build\_Model(QJsonObject &jsonObject, QHash<uint, node> &Node, QHash<uint,edge> &Edge): Строит модель из JSON-данных.
  + int culc\_Eout(): Выполняет расчет поля рассеяния.
  + cVect getEout(size\_t iX, size\_t iY, size\_t iZ): Возвращает значение поля на заданной позиции.
  + void setEout(size\_t iX, size\_t iY, size\_t iZ, cVect Eout): Устанавливает значение поля на заданной позиции.
* Управление Параметрами
  + void set\_Lmax(double L), void set\_stepXYZ(double x, double y, double z), void set\_boolXYZ(bool X, bool Y, bool Z), и др.: Методы для настройки параметров вычислений.

Особенности:

* Использует вспомогательные функции для построения матриц вращения и выполнения FFT.
* Позволяет сохранять результаты в файлы в различных форматах.

## 3.4 Класс radarCore

Расположение: radar\_core.cpp

Описание: Класс, представляющий вычислительное ядро для обработки данных радиопортрета. Работает в отдельном потоке, выполняет вычисления и взаимодействует с клиентом.

Основные Члены:

* QJsonDocument m\_doc: Входные данные в формате JSON.
* clientAI \*m\_clientRadar: Указатель на клиента, инициировавшего задачу.
* QWebSocket \*m\_Client: WebSocket-соединение с клиентом.
* bool RUN: Флаг управления выполнением задачи.
* bool TEST: Флаг режима тестирования.
* int id: Идентификатор задачи.
* culcradar radar: Объект для выполнения расчетов.

Основные Методы:

* Деструктор
  + ~radarCore(): Останавливает задачу и освобождает ресурсы.
* Запуск Задачи
  + void run(): Основной метод выполнения задачи. Выполняет процесс расчета, обрабатывает исключения и отправляет результаты клиенту.
* Управление Задачей
  + void stop(): Останавливает выполнение задачи.
  + void pause\_core(): Приостанавливает выполнение задачи.
  + void continue\_core(): Возобновляет выполнение задачи.
  + void testing(bool test): Включает или выключает режим тестирования.
* Обработка Входных Данных
  + void setRadarParam(QJsonDocument &doc, clientAI \*client, QWebSocket \*pClient): Устанавливает параметры задачи.
  + void parseJSONtoRadar(QHash<uint, node> &Node, QHash<uint,edge> &Edge): Парсит JSON-данные и строит модель.
* Выполнение Вычислений
  + void calcRadar(): Выполняет расчеты радиопортрета.
  + void calcRadarResult(): Формирует и отправляет результаты клиенту.
* Отправка Сообщений Клиенту
  + void sendText(): Отправляет текстовое сообщение клиенту.
  + void sendProgressBar(): Отправляет обновления прогресса клиенту.

Особенности:

* Работает в отдельном потоке для предотвращения блокировки основного потока сервера.
* Использует сигналы и слоты для асинхронной коммуникации с сервером и клиентом.

## 3.5 Утилиты calctools

Расположение: calctools.cpp

Описание: Вспомогательные функции для логирования и отправки ответов клиенту.

Основные Функции:

* void clogs(QString msg, QString type, QString Source): Логирует сообщения в файл и стандартный вывод.
  + msg: Сообщение для логирования.
  + type: Тип сообщения ("err", "wrn", или пустая строка для информационных сообщений).
  + Source: Источник сообщения (по умолчанию "сервер").
* void webServerAnswer(QString answer, QWebSocket \*pClient): Отправляет ответное сообщение клиенту в формате JSON.
  + answer: Текст ответа.
  + pClient: WebSocket-соединение клиента.

# 4 ПРОТОКОЛ ВЗАИМОДЕЙСТВИЯ

## 4.1 Формат Сообщений

Взаимодействие между клиентом и сервером осуществляется посредством обмена сообщениями в формате JSON через WebSocket. Каждое сообщение содержит поле "type", определяющее тип сообщения, и дополнительные поля в зависимости от типа.

Основные Типы Сообщений:

* Аутентификация ("auth")

|  |
| --- |
| {  "type": "auth",  "login": "username",  "password": "password"  } |

* Исходные Данные ("triangles")

|  |
| --- |
| {  "type": "triangles",  "data": {  "0": {"x": -16.539032, "y": -9.142952, "z": 44.991859},  "1": {"x": 20.046467, "y": -20.242254, "z": 33.23822},  // ... другие вершины ...  },  "visibleTriangles": [true, false, false, true, true, true, false, true, true, false, true, true],  "freqBand": 2,  "polarRadiation": 1,  "polarRecive": 1,  "resolution": 1,  "typeAngle": true,  "typeAzimut": false,  "typeLength": true,  "pplane": true,  "directVector": {"x": -0.13109064102172852, "y": 5.009283065795898, "z": -185.3519744873047},  "wavelength": 1  } |

* + data: Основной объект, содержащий координаты треугольников. Каждый треугольник определяется тремя вершинами, каждая из которых представлена тремя координатами (x, y, z).
  + visibleTriangles: Массив булевых значений, указывающих, виден ли соответствующий треугольник.
  + freqBand: Целое число, указывающее частотную полосу.
  + polarRadiation: Целое число, указывающее поляризацию излучения.
  + polarRecive: Целое число, указывающее поляризацию приёма.
  + typeAngle: Булево значение, указывающее, используется ли угловое изображение по высоте.
  + typeAzimut: Булево значение, указывающее, используется ли угловое изображение по азимуту.
  + typeLength: Булево значение, указывающее, используется ли изображение по дальности.
  + pplane: Булево значение, указывающее учет подстилающей поверхности.
  + directVector: Объект, содержащий координаты направляющего вектора (x, y, z).
  + wavelength: Длина волны.
* Команды ("cmd")

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["status"]  } |

* Ответы Сервером ("answer", "progress\_bar", "result")

|  |
| --- |
| {  "type": "answer",  "msg": "сообщение"  } |

|  |
| --- |
| {  "type": "progress\_bar",  "id": 1,  "status": "work",  "content": 50  } |

|  |
| --- |
| {  "type": "result",  "content": {...}  } |

## 4.2 Этапы Взаимодействия

1. Установление Соединения
   * Клиент инициирует подключение к серверу через WebSocket на заданном порту.
   * Сервер принимает подключение, генерирует уникальный идентификатор для клиента и отправляет подтверждение.
2. Аутентификация
   * Клиент отправляет сообщение типа "auth" с логином и паролем.
   * Сервер проверяет учетные данные, используя файл users.json (если он доступен) или предопределенные значения ("user"/"user").
   * В случае успешной аутентификации сервер отправляет сообщение "клиент подключен", устанавливает статус аутентификации для клиента.
3. Отправка Исходных Данных
   * После аутентификации клиент отправляет сообщение типа "triangles" с данными для построения радиопортрета.
   * Сервер обрабатывает данные, проверяет наличие существующих задач для данного клиента, инициирует новую вычислительную задачу, создавая объект radarCore и запускает его в отдельном потоке.
4. Выполнение Вычислений
   * radarCore выполняет расчеты, периодически отправляя обновления прогресса через сообщения типа "progress\_bar".
   * По завершении вычислений сервер отправляет результаты клиенту через сообщение типа "result".
5. Отправка Команд Управления
   * Клиент может отправлять команды типа "cmd" для управления задачами:
     + "status": Запрос статуса сервера.
     + "whoconnect": Получение списка подключенных клиентов.
     + "pause", "continue", "stop": Управление выполнением задач.
     + Дополнительные команды для сохранения данных, тестирования и передачи моделей.
6. Закрытие Соединения
   * Клиент может отключиться от сервера в любое время.
   * Сервер обрабатывает отключение, освобождает ресурсы и завершает связанные задачи.

## 4.3 Клиентская сторона взаимодействия

Клиентская сторона реализована в классе TriangleClient. Ниже приведены основные методы и события, связанные с взаимодействием:

4.3.1 Инициализация и подключение

Клиент инициализирует объект WebSocket, устанавливает необходимые соединения для обработки событий и открывает соединение с сервером. При подключении устанавливаются обработчики для следующих событий:

* Подключение к серверу (onConnected)
* Отключение от сервера (onDisconnected)
* Получение сообщений от сервера (onTextMessageReceived)
* Обработка ошибок соединения (onErrorOccurred)

### 4.3.2 Отправка данных

Процесс отправки данных о треугольниках на сервер включает следующие шаги:

* Подготовка данных:
  + Собираются данные о треугольниках, включая их координаты и видимость.
  + Формируется JSON-объект с необходимыми параметрами, такими как type, data, polarRadiation, polarRecive, typeAngle, typeAzimut, typeLength.
* Отправка сообщения:
  + JSON-объект преобразуется в строку и отправляется на сервер через WebSocket.
  + В случае ошибки отправки осуществляется попытка повторной отправки через заданный интервал времени.
    1. Обработка полученных сообщений
* Логирование
  + Асинхронное логирование реализовано через LogWorker и поток m\_logThread для обеспечения непрерывной работы клиента без блокировок.

### 4.3.4 Разбор входящих JSON файлов

Процесс разбора JSON включает следующие шаги:

1. Преобразование строки в JSON:
   * Сообщение, полученное в виде строки, преобразуется в формат JSON.
2. Проверка корректности JSON:
   * Если JSON содержит ошибки, логируется сообщение об ошибке, и дальнейшая обработка этого сообщения прекращается.
3. Проверка структуры JSON:
   * Проверяется, является ли разобранный JSON объектом и содержит ли он необходимые поля, такие как type.
4. Обработка содержимого JSON:
   * В зависимости от значения поля type, клиент выполняет соответствующие действия:
     + result: Обработка результатов, отправленных сервером, и выполнение необходимых действий с данными.
     + answer: Обработка ответов сервера, таких как подтверждения успешной аутентификации.
     + progress\_bar: Обновление прогресс-бара с учетом полученного значения прогресса.

4.3.5 Авторизация и отправка команд

Для авторизации клиента на сервере используется метод authorize, который отправляет логин и пароль в формате JSON. Также клиент может отправлять различные команды на сервер через метод sendCommand, формируя соответствующий JSON-объект с командой и параметрами.

# 5 УПРАВЛЕНИЕ ЗАДАЧАМИ И ПОТОКАМИ

5.1 Инициация Вычислительной Задачи

Когда сервер получает исходные данные от клиента ("triangles"), он выполняет следующие шаги:

* + 1. Проверка Наличия Существующих Задач:
  + Сервер проверяет, не запущена ли уже задача с тем же идентификатором модели для данного клиента.
  + Если задача уже запущена, сервер уведомляет клиента и не создает новую задачу.

5.1.2 Создание Потока и Ядра Вычислений:

* + Сервер создает новый объект QThread для выполнения задачи.
  + Создает объект radarCore, перемещает его в созданный поток.
  + Устанавливает связи между сигналами и слотами для управления жизненным циклом потока и получения результатов.

5.1.3 Настройка Параметров:

* + Передает параметры задачи (QJsonDocument, clientAI, QWebSocket) объекту radarCore.
    1. Запуск Потока:
  + Запускает поток, который вызывает метод run() объекта radarCore.

5.1.5 Добавление Задачи в Список:

* + Добавляет объект radarCore в список активных задач task\_list.

5.2 Завершение Задачи

Задача может завершиться по нескольким причинам:

* Успешное Завершение:
  + После выполнения всех расчетов и отправки результатов, radarCore сигнализирует о завершении через сигнал finished().
  + Сервер удаляет объект задачи из списка task\_list и освобождает ресурсы.
* Принудительное Завершение:
  + Клиент может отправить команду "stop", после чего сервер вызывает метод task\_kill(), который останавливает и удаляет соответствующую задачу.
  + Если задача прерывается по причине ошибки или непредвиденного события, сервер также освобождает ресурсы.

5.3 Управление Потоками

Каждая задача выполняется в отдельном потоке (QThread), что позволяет серверу обрабатывать несколько клиентов и задач одновременно без блокировки основного потока.

Сигналы и Слоты:

* Сигналы от radarCore:
  + finished(): Сигнализирует о завершении задачи.
  + send\_text(QString, QWebSocket\*): Отправляет текстовые сообщения клиенту.
  + send\_progress\_bar(QString, QWebSocket\*): Отправляет обновления прогресса.
  + send\_result(QString, QWebSocket\*): Отправляет результаты вычислений.
* Слоты от WebServer:
  + kill\_task(QWebSocket\*): Обрабатывает запрос на завершение задачи.
  + message\_calc\_radar(QString, QWebSocket\*): Отправляет текстовые сообщения клиенту.
  + send\_calc\_radar\_result(QString, QWebSocket\*): Отправляет результаты клиенту.

# 6 ОБРАБОТКА КОМАНД

Сервер поддерживает различные команды, отправляемые клиентами, для управления сервером и задачами. Все команды приходят в сообщениях типа "cmd".

## 6.1 Поддерживаемые Команды

* 1. .1 "server": Команды управления сервером.
  + "status": Получение статуса сервера.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["status"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "Статус: [OK] Версия: 1.0 Клиент ID: <id> Плагины: #include <Radar Core> АО 'НЦ ПЭ' 2022, все права защищены"  } |

* + "whoconnect": Получение списка подключенных клиентов.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["whoconnect"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "Подключены клиенты: 0. user1 [id1] <12.00.00 01.01.2023> 1. user2 [id2] <13.30.00 02.01.2023> ..."  } |

* + "pause": Поставить выполнение задачи на паузу.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["pause"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "пауза для: <n>. user [id] Статус: [OK] приостановка вычислений"  } |

* + "continue": Возобновить выполнение задачи.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["continue"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "продолжение вычислений для: <n>. user [id] Статус: [OK] продолжение вычислений"  } |

* + "stop": Принудительно завершить выполнение задачи.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["stop"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "принудительное завершение вычислений для: <n>. user [id] Статус: [OK] ресурс освобожден"  } |

* + "start\_test": Включить режим тестирования.

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["start\_test"]  } |

Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "перевод в режим тестирования для: <n>. user [id] Статус: [OK] режим тестирования для [id] включен"  } |

* + "save\_file\_on", "save\_model\_on", "save\_scat\_on", "save\_fft\_on", "serialize\_model\_on", "read\_model\_on", "send\_model", "read\_result", "save\_message": Управление сохранением и загрузкой данных.
    - Пример Команды:

|  |
| --- |
| {  "type": "cmd",  "cmd": "server",  "param": ["save\_model\_on"]  } |

* + - Ответ:

|  |
| --- |
| {  "type": "answer",  "msg": "запись модели в файл включена"  } |

### 6.1.2 Неизвестные Команды

* + Если сервер получает неизвестную команду, он отправляет сообщение об ошибке.

|  |
| --- |
| {  "type": "answer",  "msg": "команда не зарегистрирована"  } |

6.2 Метод handlerCmd(QString command, QJsonArray \*params, QWebSocket \*pSender) в классе WebServer отвечает за обработку команд. В зависимости от команды и параметров, сервер выполняет соответствующие действия:

* "status": Отправляет статус сервера.
* "whoconnect": Формирует список подключенных клиентов и отправляет его.
* "pause", "continue", "stop": Управляют выполнением задач через методы pause\_core(), continue\_core(), stop().
* "start\_test": Инициирует режим тестирования, возможно, с использованием файла testing.json.
* Флаги сохранения: Включают соответствующие флаги (SAVE\_MODEL, SAVE\_SCAT\_FIELD и др.), которые используются при обработке входных данных для сохранения результатов в файлы.

6.3 Метод setCmd(QString &message, QWebSocket \*pSender, int &cmd\_id) генерирует и отправляет ответы на команды, а также выполняет необходимые действия над задачами.

# 7 РАБОТА С ФАЙЛАМИ

Сервер взаимодействует с файловой системой для различных целей, включая тестирование, сохранение входных данных и результатов, а также загрузку моделей.

* 1. Файлы Используемые Сервером

7.1.1 users.json: Содержит учетные данные клиентов для аутентификации.

* + Формат:

|  |
| --- |
| {  "user1": "password1",  "user2": "password2",  ...  } |

* + 1. testing.json: Используется для включения режима тестирования. Если файл существует и содержит определенные команды, сервер работает в тестовом режиме.

7.1.3 message.bin: Бинарный файл для записи сообщений от клиентов.

7.1.4 input\_data.bin и input\_data.json: Файлы для сохранения входных данных моделей.

* + 1. model.json: Файл для сохранения модели в формате JSON.

7.1.6 scat\_field.json и fft\_field.json: Файлы для сохранения полей рассеяния и их FFT-преобразований.

* + 1. client\_message.json и client\_message.bin: Файлы для сохранения сообщений, отправленных клиенту.

## 7.2 Операции с Файлами

* Чтение Файлов
  + При аутентификации (userVerification), сервер пытается открыть users.json для проверки учетных данных.
  + При обработке команд тестирования ("start\_test"), сервер читает testing.json.
* Запись Файлов
  + В зависимости от установленных флагов (SAVE\_MODEL, SAVE\_SCAT\_FIELD и др.), сервер записывает данные в соответствующие файлы.
  + Например, при установке SAVE\_MODEL, сервер сохраняет модель в model.json.
* Обработка Ошибок При Открытии Файлов
  + Если файл не может быть открыт для чтения или записи, сервер логирует ошибку и уведомляет клиента.

## 7.3 Примеры Операций

7.3.1 Чтение users.json:

|  |
| --- |
| void WebServer::userVerification(QString login, QString password, QWebSocket \*pSender) {  QFile file0("users.json");  if (file0.open(QIODevice::ReadOnly | QIODevice::Text)) {  QByteArray auth\_data = file0.readAll();  QJsonDocument auth\_json = QJsonDocument::fromJson(auth\_data.toUtf8());  QJsonObject jsonObject = auth\_json.object();  if (jsonObject.contains(login) && password == jsonObject.value(login).toString()) {  // Успешная аутентификация  } else {  // Неудачная аутентификация  }  file0.close();  } else {  // Использование предопределенных учетных данных  }  } |

7.3.2 Запись model.json:

|  |
| --- |
| if (SAVE\_MODEL\_TO\_FILE) {  QFile file("model.json");  QJsonDocument Model(model\_obj);  if(!file.open(QIODevice::WriteOnly)) {  clogs("file model.json error to open","","");  }  else {  clogs("save file model.json","","");  file.resize(0);  file.write(Model.toJson());  }  file.close();  SAVE\_MODEL\_TO\_FILE = false;  } |

# 8 ОБРАБОТКА ОШИБОК

Сервер реализует несколько уровней обработки ошибок для обеспечения надежности и стабильности.

* 1. Виды Ошибок

8.1.1 Ошибки Аутентификации

* + Неверный логин или пароль.
  + Отсутствие файла users.json, что приводит к использованию предопределенных учетных данных.
    1. Ошибки Входных Данных
  + Отсутствие необходимых полей в JSON-сообщении (например, "data", "visibleTriangles", "directVector").
  + Некорректные данные (например, количество координат не кратно 9).

8.1.3 Ошибки Файловой Системы

* + Невозможность открытия файлов для чтения или записи.
  + Пустые файлы testing.json или users.json.

8.1.4 Внутренние Ошибки Вычислений

* + Ошибки при установке поляризации.
  + Прерывание вычислений пользователем или из-за нехватки памяти (std::bad\_alloc).

## 8.2 Механизмы Обработки Ошибок

* Логирование: Все ошибки логируются с помощью функции clogs, которая записывает сообщения в лог-файл и выводит их в стандартный вывод.

clogs("ошибка открытия файла message.bin для записи", "err", "");

* Отправка Сообщений Клиенту: При возникновении ошибки сервер отправляет клиенту соответствующее сообщение типа "answer".

webServerAnswer("принятое сообщение имеет неизвестный тип", pSender);

* Исключения: В методе run() класса radarCore обрабатываются исключения, такие как std::bad\_alloc или пользовательские коды ошибок, для безопасного завершения задачи и информирования клиента.

|  |
| --- |
| catch (int a) {  if (a == -1) {  Txt = "ядро остановлено"; sendText();  emit finished();  emit task\_kill(m\_Client);  }  }  catch (std::bad\_alloc &ba) {  Txt = "нехватка памяти"; sendText();  emit finished();  emit task\_kill(m\_Client);  } |

## 8.3 Примеры Обработки Ошибок

8.3.1 Ошибка Отсутствия Поля "data":

|  |
| --- |
| if (!jsonObject.contains("data")) {  qDebug() << "Error: 'data' not found in JSON";  return 1; // Код ошибки  } |

8.3.2 Ошибка При Аутентификации:

|  |
| --- |
| else {  answer = "клиент не авторизован";  clogs("клиент [" + login + "] не авторизован", "", "");  webServerAnswer(answer, pSender);  } |

# 9 ЛОГИРОВАНИЕ

Логирование осуществляется через функцию clogs, которая записывает сообщения как в лог-файл, так и в стандартный вывод (консоль).

## 9.1 Формат Логирования

|  |
| --- |
| void clogs(QString msg, QString type, QString Source) {  QDate cd = QDate::currentDate();  QTime ct = QTime::currentTime();  QString typeMsg = "";  if (type == "err")  typeMsg = "<ERROR> ";  if (type == "wrn")  typeMsg = "<WARNING> ";  if (Source == "")  Source = "сервер";  Source = " (" + Source + ") ";  QString logmsg = "[" + cd.toString("dd.MM.yyyy") + " " +  ct.toString("hh.mm.ss") + "] " + typeMsg + Source + msg +  "\n";  QTextStream out(m\_logFile.data());  out << logmsg;  out.flush();  QTextStream(stdout) << logmsg;  } |

## 9.2 Параметры:

* msg: Сообщение для логирования.
* type: Тип сообщения ("err", "wrn", или пустая строка для информационных сообщений).
* Source: Источник сообщения (по умолчанию "сервер").

## 9.3 Примеры Использования:

* clogs("старт сервера", "", "");
* clogs("ошибка открытия файла message.bin для записи", "err", "");
* clogs("новый сеанс подключения [" + newClient->id + "]", "", "");