МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___» _____ 2025 г.

Разработка OPC UA клиента для считывания содержимого регистров программируемого логического контроллера

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУРГУ-090301.2025.405 ПЗ ВКР

Руководитель работы,
к.т.н., доцент каф. ЭВМ
_____ Д.В. Топольский
«___» _____2025 г.

Автор работы,
студент группы КЭ-405
_____А.С. Вовк
«___» _____2025 г.

Нормоконтролёр,
ст. преп. каф. ЭВМ
_____ С.В. Сяськов
«___» _____2025 г.

Челябинск-2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___»_____2025 г.

ЗАДАНИЕ

на выпускную квалификационную работу бакалавра

студенту группы КЭ-405
Вовку Андрею Сергеевичу
обучающемуся по направлению
09.03.01 «Информатика и вычислительная техника»

1. **Тема работы: «**Разработка OPC UA клиента для считывания содержимого регистров программируемого логического контроллера» утверждена приказом по университету от 21 апреля 2025 г. № 648-13/12.

2. **Срок сдачи студентом законченной работы:** 01 июня 2025 г.

3. **Исходные данные к работе:**
   1. Разработка OPC UA клиента, характеристики:
      - сетевой протокол: OPC UA версии 1.04;
      - тип подключения: TCP/IP;
      - считывание и запись данных;
      - поддержка анонимного подключения и авторизации через логин/пароль;
      - шифрование данных.

   Интеграция:

- совместимость с Modbus RTU/TCP;
- взаимодействие с SCADA системами.

2. Нефункциональные требования:

   Использование стандарта OPC Unified Architecture (IEC 62541).

   2.1. Минимальные системные требования для клиента:

   - операционная система: Windows 10 или Linux;
   - процессор: Intel i5-4590 / AMD Ryzen 5 1500X или лучше;
   - оперативная память: не менее 8 ГБ;
   - сетевые интерфейсы: Ethernet или Wi-Fi (5 ГГц);
   - программное обеспечение: поддержка OpenSSL для шифрования.

4. Перечень подлежащих разработке вопросов:

1. Аналитический обзор научно-технической, нормативной и методической литературы по тематике работы.
2. Архитектура стандарта OPC UA для передачи данных в промышленных сетях и проектирование клиента.
3. Программная реализация приложения OPC UA клиента для считывания содержимого регистров с устройств ввода/вывода программируемого логического контроллера.
4. Тестирование разработанного приложения.

5. **Дата выдачи задания:** 2 декабря 2024 года

Руководитель работы _____/ *Д.В. Топольский*/

Студент _____/ *А.С. Вовк* /

КАЛЕНДАРНЫЙ ПЛАН

Этап	Срок сдачи	Подпись руководителя
Введение и обзор литературы	03.03.2025	
Архитектура и проектирование приложения OPC UA	22.03.2025	
Программная реализация приложения клиента	12.04.2025	
Тестирование приложения	26.04.2025	
Компоновка текста работы и сдача на нормоконтроль	22.05.2025	
Подготовка презентации и доклада	30.05.2025	

Руководитель работы _____/ *Д.В. Топольский* /

Студент _____/ *А.С. Вовк* /

Аннотация

А.С. Вовк. Разработка OPC UA клиента для считывания содержимого регистров программируемого логического контроллера. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШ ЭКН; 2025, 103 с., 18 ил., библиогр. список – 24 наим.

В рамках данной выпускной квалификационной работы был проведён детальный анализ современных научных и технических источников, посвящённых созданию OPC UA клиента для считывания данных из регистров программируемого логического контроллера (ПЛК). Особое внимание уделялось архитектуре и функциональным особенностям OPC UA, включая специфику самого протокола и его преимущества по сравнению с альтернативными протоколами передачи данных. Были рассмотрены различные способы интеграции клиента с ПЛК, приведены примеры реализации на ряде языков программирования. Кроме того, проанализированы вопросы, связанные с безопасностью и надёжностью при взаимодействии устройств по протоколу OPC UA. В результате выполненной работы удалось сформулировать ключевые принципы функционирования современных OPC UA клиентов, обеспечивающих эффективный и безопасный обмен данными с разнообразным промышленным оборудование

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 8
1. АНАЛИТИЧЕСКИЙ ОБЗОР СОВРЕМЕННОЙ НАУЧНО-ТЕХНИЧЕСКОЙ, НОРМАТИВНОЙ, МЕТОДИЧЕСКОЙ ЛИТЕРАТУРЫ, ЗАТРАГИВАЮЩЕЙ ИССЛЕДУЕМУЮ НАУЧНО-ТЕХНИЧЕСКУЮ ПРОБЛЕМУ 9
ВЫВОД К ГЛАВЕ 1 26
2.АРХИТЕКТУРА И ПРОЕКТИРОВАНИЕ OPC UA 27
2.1. ЦЕЛИ ПРОЕКТИРОВАНИЯ OPC UA 27
2.2. АРХИТЕКТУРА OPC UA СИСТЕМЫ 30
2.3. АРХИТЕКТУРА OPC UA КЛИЕНТА 30
2.4. АРХИТЕКТУРА OPC UA СЕРВЕРА 32
2.5. АДРЕСНОЕ ПРОСТРАНСТВО OPC UA 33
2.6. АНАЛИЗ ПРОЕКТИРУЕМОЙ СИСТЕМЫ 34
2.6.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 35
2.6.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ 36
2.7. ПРОЕКТИРОВАНИЕ 37
ВЫВОД К ГЛАВЕ 2 39
3. РЕАЛИЗАЦИЯ 40
3.1. СТЕК ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ 40
3.2. РАЗРАБОТКА БАЗЫ ДАННЫХ 42
3.3. ОПИСАНИЕ ИСПОЛЬЗУЕМЫХ БИБЛИОТЕК 45
3.4. ОПИСАНИЕ РЕАЛИЗАЦИИ И АЛГОРИТМА РАБОТЫ ПРОГРАММЫ 46
ВЫВОД К ГЛАВЕ 3 50
4. ТЕСТИРОВАНИЕ 51
4.1. ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ 51
4.2. НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ 53
ВЫВОД ПО ГЛАВЕ 4 54

ЗАКЛЮЧЕНИЕ 55
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 56
ПРИЛОЖЕНИЕ А МОДУЛЬ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ POSTGRESQL 59
ПРИЛОЖЕНИЕ Б МОДУЛЬ OPC UA КЛИЕНТА 66
ПРИЛОЖЕНИЕ В РЕАЛИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА 80
ПРИЛОЖЕНИЕ Г ИНТЕГРИРОВАННЫЕ МОДЕЛИ И СЕРВИСЫ OPC UA .. 98

ВВЕДЕНИЕ

Разработка OPC UA клиента, предназначенного для считывания содержимого регистров программируемого логического контроллера (ПЛК), представляет собой важный этап в эволюции систем автоматизации управления технологическими процессами. Стандарт OPC UA (Open Platform Communications Unified Architecture) сегодня является одним из наиболее актуальных решений, обеспечивающих безопасную и эффективную передачу данных между различными устройствами и программным обеспечением.

Клиенты, построенные на базе этого протокола, позволяют интегрировать ПЛК с другими компонентами автоматизированных систем, обеспечивая удобный доступ к данным. Информация, получаемая от контроллера, преобразуется в формат, подходящий для анализа, визуализации и дальнейшей обработки, что даёт возможность применять универсальные инструменты мониторинга и управления, независимо от используемого оборудования.

Передача данных осуществляется по защищённым каналам, что гарантирует их безопасность и целостность. При разработке OPC UA клиента учитываются архитектурные особенности программного решения, выбор подходящего языка программирования, а также меры по защите информации. Компетентная реализация такого клиента способствует повышению надёжности и эффективности автоматизированных систем управления.

1. АНАЛИТИЧЕСКИЙ ОБЗОР СОВРЕМЕННОЙ НАУЧНО-ТЕХНИЧЕСКОЙ, НОРМАТИВНОЙ, МЕТОДИЧЕСКОЙ ЛИТЕРАТУРЫ, ЗАТРАГИВАЮЩЕЙ ИССЛЕДУЕМУЮ НАУЧНО-ТЕХНИЧЕСКУЮ ПРОБЛЕМУ

Стандарт OPC UA (Open Platform Communications Unified Architecture) представляет собой универсальный инструмент для передачи данных в системах промышленной автоматизации. Его архитектура включает ряд ключевых компонентов, каждый из которых играет свою роль в обеспечении надёжности, безопасности и гибкости при работе с информацией.

Ключевым элементом архитектуры является сервер OPC UA, предоставляющий доступ к данным и функциональным сервисам. Он может работать на различных платформах и обслуживать множество клиентов одновременно, что делает его удобным для интеграции в существующие автоматизированные системы. Такие серверы способны выполнять запросы на чтение и запись данных, а также предоставлять сведения о состоянии оборудования, обеспечивая пользователей актуальной информацией о ходе производственных процессов [1].

OPC UA поддерживает разные модели взаимодействия – как классический клиент-сервер, так и модель подписки. В первом случае клиент обращается к серверу при необходимости, во втором – сервер сам уведомляет клиента об изменениях, снижая сетевую нагрузку и повышая эффективность обмена [6].

Клиенты OPC UA инициируют запросы к серверу, выполняя такие операции, как чтение и запись переменных, вызов методов и подписка на события. Они могут быть реализованы на различных языках программирования и платформах, от настольных до мобильных устройств, что обеспечивает высокую гибкость их применения [2].

Для повышения производительности клиенты могут использовать механизмы кэширования часто используемых данных. Это позволяет сократить количество запросов к серверу и ускорить доступ к информации. Также, клиенты

должны поддерживать механизмы безопасности для защиты передачи данных [1].

Основой для организации данных и их взаимосвязей является информационная модель OPC UA, которая позволяет структурировать данные в виде объектов, переменных и методов. Такая структура упрощает управление информацией и её использованием в приложениях. Информационная модель поддерживает создание сложных иерархий данных, что необходимо для промышленных приложений [16].

Каждый объект в информационной модели может содержать атрибуты (значения переменных) и методы (функции, которые могут быть вызваны). Это позволяет разработчикам создавать более интуитивно понятные интерфейсы для взаимодействия с данными. Также, информационная модель поддерживает расширяемость, которая позволяет разработчика добавлять новые типы узлов и атрибутов без изменения существующей структуры [6]. OPC UA использует протоколы передачи данных, включая TCP/IP для обмена данными в реальном времени, HTTP/HTTPS для работы через веб-приложения и WebSockets для реализации двустороннего обмена данными. Такое разнообразие протоколов делает OPC UA универсальным стандартом для интеграции различных систем и устройств [24].

Протокол передачи данных обеспечивает надежность и безопасность обмена информацией между клиентами и серверами. Использование шифрования данных при передаче защищает информацию от несанкционированного доступа. Также важным аспектом является возможность настройки параметров связи для оптимизации производительности системы [1]

Безопасность является одним из базовых компонентов архитектуры OPC UA. Стандарт предусматривает использование шифрования данных, аутентификацию пользователей и управление доступом к информации, что обеспечивает защиту от несанкционированного доступа и атак. Это особенно критично в промышленных системах [1]

OPC UA использует многоуровневую аутентификацию с применением сертификатов, позволяя гарантировать идентификацию пользователей и устройств перед началом обмена данными. Также, стандарт поддерживает управление правами доступа к данным на уровне отдельных узлов информационной модели.

Адресное пространство OPC UA представляет собой структурированную совокупность всех узлов в системе. Каждый узел имеет уникальный идентификатор и может содержать атрибуты и методы, что позволяет клиентам эффективно находить и использовать нужные данные для взаимодействия с ними [16].

Адресное пространство организовано в виде дерева, где корневой узел представляет собой сервер, а дочерние узлы – различные объекты и переменные системы. Такая структура упрощает навигацию по данным, делая взаимодействие с ними более интуитивным.

OPC UA поддерживает механизм подписки на изменения значений переменных или события в системе. Это позволяет клиентам получать уведомления о произошедших изменениях без необходимости постоянного опроса сервера. Такой подход значительно снижает нагрузку на сеть и сервер [6].

На сегодняшний день существуют две основные спецификации технологии Open Platform Communication (OPC) – Classic [15], которая включает в себя OPC Data Access (OPC DA), OPC Historical Data Access (OPC HDA), OPC Alarms & Events (OPC AE) и OPC Unified Architecture (OPC UA) [16]. OPC Classic использует технологию DCOM в качестве своей основы [17].

DCOM (Distributed Component Object Model), или распределенная модель компонентных объектов, является расширением модели COM, разработанной компанией Microsoft. Это расширение ориентировано на поддержку и интеграцию функционирующих в сети распределенных объектных приложений.

Ключевым аспектом COM является обеспечение связи между клиентами и серверами посредством интерфейсов. Именно интерфейс предоставляет клиенту способ определить поддерживаемые на этапе выполнения технологии сервера.

Для расширения возможностей сервера достаточно добавить новый интерфейс к существующим.

Несмотря на то, что OPC Classic приобрел огромный успех, данная спецификация считается устаревшей и имеет следующие недостатки:

- доступность данной технологии только на операционных системах семейства Windows;
- связь с технологией DCOM. Исходные коды данной технологии являются закрытыми, что не позволяет решать вопросы надежности программного обеспечения. Это также затрудняет выявление и устранение возникающих программных сбоев;
- возникновение трудностей конфигурирования, связанных с DCOM;
- неточные сообщения DCOM о прерываниях связи;
- неприспособленность DCOM для обмена данными через интернет;
- неприспособленность DCOM для обеспечения информационной безопасности.

В связи с вышеуказанными недостатками консорциумом OPC Foundation было принято решение разработать новую спецификацию, свободную от этих недостатков. Новая спецификация получила унифицированную архитектуру и название OPC UA. Основные ее достоинства:

- реализация на языке программирования ANSI C для обеспечения переносимости на другие платформы, включая встраиваемые системы. Также доступны версии на .NET и Java, разработанные OPC Foundation;
- данная спецификация ориентирована на сервисы, а не на объекты, что позволяет использовать спецификацию OPC UA на любых устройствах, использующих веб-сервисы; − возможность масштабирования OPC UA, то есть изменение объема программы в зависимости от вычислительных ресурсов процессора и требуемой функциональности;

Также может быть выполнена компиляция в виде однопоточного или многопоточного приложения; − поддержка надежного и современного транспортного механизма SOAP (Simple Object Access Protocol) на базе XML (eXtensible Markup Language) с применением протокола HTTP;

- обеспечение высокой степени информационной безопасности;
- конфигурируемый таймаут для каждого сервиса;
- использование открытых стандартов World Wide Web Consortium (W3C) вместо закрытого стандарта COM/DCOM.

Стоит отметить, что на современных контроллерах установлена операционная система на базе ядра Linux, что позволяет запускать на них сервер OPC UA.

Таблица 1 – Сравнительная таблица стандартов OPC

Стандарт OPC	**Описание**	**Основные особенности**
OPC DA (Data Access)	Обеспечивает обмен данными в реальном времени с устройствами, такими как ПЛК и ЧМИ.	Наиболее распространённый стандарт, поддерживает синхронный и асинхронный обмен данными.
OPC HAD (Historical Data Access)	Предоставляет доступ к сохранённым историческим данным.	Позволяет работать с архивами данных, в отличие от OPC DA, который фокусируется на текущих данных.
OPC AE (Alarms & Events)	Уведомляет о событиях, таких как аварии и действия оператора.	Поддерживает уведомления по запросу и автоматические оповещения о событиях.
OPC Batch	Управляет процессами на основе рецептур и шагов.	Применяется для управления технологическими процессами, соответствующими стандарту S88.01.
OPC DX (Data eXchange)	Обеспечивает обмен данными между OPC-серверами через Ethernet.	Создаёт шлюзы для взаимодействия между устройствами разных производителей.

Продолжение таблицы 1

Стандарт OPC	Описание	Основные особенности
OPC Security	Определяет управление доступом клиентов к данным OPC-сервера.	Включает функции аутентификации и авторизации для защиты данных.
OPC UA (Unified Architecture)	Унифицированный стандарт, обеспечивающий кроссплатформенную совместимость.	Поддерживает богатую информационную модель, безопасность данных и работу через интернет-протоколы, такие как TCP и HTTP/SOAP.

Таким образом исходя из вышеперечисленных данных наиболее релятивным стандартом OPC для создания клиента, считывающего содержимое регистров ПЛК является OPC UA (Unified Architecture).

Кроме того, ряд производителей предлагает контроллеры с уже установленным сервером OPC UA. Например, контроллеры производства Овен ПЛК200, Siemens S7-1500 и другие. Сервер OPC UA конфигурируется во время разработки проекта, после чего его можно опросить с помощью разработанного клиента. Разрабатываемое программное обеспечение создается с учетом того, что оно сможет запускаться на любой современной операционной системе так же, как и сервер OPC UA.

Стандарт OPC UA демонстрирует высокую универсальность и эффективность при применении в различных отраслях, обеспечивая надёжный и безопасный обмен данными между оборудованием и информационными системами. Проведённый анализ успешных кейсов внедрения показывает, как использование OPC UA способствует повышению интеграции, надёжности и эффективности в таких сферах, как промышленность, энергетика, транспорт и здравоохранение.

В производственной сфере OPC UA широко применяется для объединения различных систем автоматизации. К примеру, на предприятиях по выпуску

электронной продукции данный стандарт используется для интеграции данных от различных участков и устройств, что обеспечивает централизованное управление технологическими процессами. В одном из реализованных проектов на заводе была внедрена система сбора данных с применением OPC UA, что позволило значительно сократить время на диагностику неисправностей и повысить общую производительность производственной линии. В статье на Habr описан пример клиент-серверного взаимодействия между контроллерами серии S7-1500 с использованием протокола OPC UA, что подтверждает его эффективность в условиях реального производства [9].

В энергетической промышленности OPC UA широко применяется для интеграции SCADA-систем с системами управления технологическими процессами (АСУ ТП), что позволяет отслеживать учет потребляемой энергии и оптимизировать эффективное распределение ресурсов. В работе Fine Start подчеркивается значимость использования данного стандарта для повышения эффективности производственных процессов в этой отрасли [23]. Стандарт позволяет осуществить безопасную передачу данных между различными устройствами, что крайне необходимо для надежности энергоснабжения.

Также OPC UA используется в области транспорта. Стандарт используется для мониторинга состояния транспортных средств и управления логистическими процессами. Например, системы управления движением могут использовать OPC UA для обмена данными между различными транспортными средствами и инфраструктурой, что позволяет улучшить координацию между транспортными средствами и снижает количество транспортных происшествий.

В области здравоохранении стандарт OPC UA применяется для интеграции медицинских устройств и систем управления данными пациентов. Медицинские приборы имеют возможность передать данные о состоянии пациента в реальном времени через протокол OPC UA в электронные медицинские записи, обеспечивая быстрый доступ к информации о состоянии здоровья пациента и улучшая качество медицинского обслуживания.

Одним из ключевых преимуществ стандарта OPC UA является его способность обеспечивать высокий уровень защиты данных. Встроенные механизмы шифрования и аутентификации пользователей делают его особенно актуальным в таких критически важных отраслях, как здравоохранение и энергетика [1]. Благодаря этим функциям удаётся эффективно предотвращать несанкционированный доступ и кибератаки.

Стандарт активно развивается, учитывая современные технологические вызовы и потребности рынка. Одной из актуальных тенденций стала интеграция OPC UA с концепцией Интернета вещей (IoT). По мере роста количества подключённых устройств возрастает потребность в безопасном и стабильном обмене данными. Использование OPC UA в IoT-среде способствует созданию интеллектуальных систем, способных к адаптации и оптимизации производственных процессов [24].

Интеграция с облачными платформами расширяет возможности удалённого доступа и анализа больших массивов данных в реальном времени, что положительно влияет на эффективность управления [23].

Кроме того, важным направлением остаётся применение алгоритмов машинного обучения и ИИ. Такие алгоритмы могут использовать данные, полученные через OPC UA, для совершенствования и автоматизации производственных процессов.

Стандарт OPC UA продолжает эволюционировать, отвечая на современные вызовы в области кибербезопасности и защиты информации. В условиях роста цифровых угроз и необходимости обеспечения надёжной защиты данных в системах промышленной автоматизации особое значение приобретает внедрение новых механизмов безопасности.

Одним из приоритетных направлений развития стало усиление систем аутентификации и авторизации. OPC UA поддерживает многоуровневую аутентификацию, в том числе с применением цифровых сертификатов, что позволяет ограничить доступ к данным исключительно авторизованными пользователями

и устройствами. Это особенно важно в условиях постоянно растущей взаимосвязанности автоматизированных систем, уязвимых к внешним воздействиям. Аутентификация может основываться как на классических методах – логин и пароль, так и на современных подходах, включая биометрию и токены.

Помимо этого, стандарт предусматривает использование современных алгоритмов шифрования при передаче и хранении информации. Это позволяет предотвращать перехват данных и обеспечивает их целостность и конфиденциальность – ключевые параметры в защите от кибератак, особенно в условиях роста объёмов передаваемой информации.

Одной из актуальных тенденций в развитии OPC UA является интеграция с решениями в области кибербезопасности. Всё больше компаний используют инструменты мониторинга и анализа, способные работать в связке с протоколом OPC UA для выявления аномалий и потенциальных угроз. Это позволяет оперативно реагировать на инциденты безопасности и снижать риски. Особое значение приобретает взаимодействие с системами управления событиями информационной безопасности (SIEM), что способствует автоматизации процессов обнаружения и реагирования на угрозы.

Значительный вклад в обеспечение безопасности вносит развитие сопутствующих стандартов. За последние годы были разработаны дополнительные спецификации, ориентированные на защиту данных. Они включают рекомендации по безопасной настройке OPC UA-серверов и клиентов, а также лучшие практики управления доступом. Эти документы способствуют унификации подходов к безопасности среди производителей аппаратного и программного обеспечения.

Не менее важным направлением остаётся обучение персонала. С ростом числа киберугроз компании всё чаще осознают необходимость подготовки сотрудников в области информационной безопасности. Обучение работе с системами на базе OPC UA и повышение осведомлённости о возможных рисках позволяют значительно усилить общий уровень защиты. Регулярные тренинги становятся ключевым элементом стратегии кибербезопасности.

Одним из приоритетных направлений развития OPC UA является применение технологий искусственного интеллекта для повышения уровня кибербезопасности. Алгоритмы машинного обучения могут эффективно анализировать сетевой трафик, выявляя отклонения в поведении пользователей или устройств. Это позволяет своевременно обнаруживать потенциальные угрозы и оперативно на них реагировать.

Благодаря своей универсальности и способности обеспечивать надёжный обмен данными между различными системами, стандарт OPC UA получил широкое распространение в различных отраслях. Ниже приведены практические кейсы, иллюстрирующие успешное внедрение данного протокола в реальных условиях.

В промышленном производстве OPC UA активно применяется для интеграции автоматизированных систем. К примеру, на предприятиях по производству электроники он используется для объединения данных от различных машин и установок, обеспечивая централизованное управление процессами. В одном из реализованных проектов на заводе была внедрена система сбора данных на базе OPC UA, что позволило значительно сократить время диагностики и устранения неисправностей, повысив общую эффективность производственной линии.

Статья на Habr описывает клиент-серверное взаимодействие между контроллерами серии S7-1500 по протоколу OPC UA, демонстрируя практическое применение технологии в условиях промышленной автоматизации [9]. Ещё один интересный пример – использование OPC UA в автомобильной отрасли. На одном из крупных автозаводов была реализована система контроля качества, основанная на OPC UA. Система собирает данные с разных этапов сборки и анализирует их на соответствие стандартам, что позволяет оперативно устранять отклонения и снижать количество дефектов. Такая интеграция обеспечивает тесную связь между производственными линиями и системами контроля качества.

В энергетике стандарт также широко применяется. Множество компаний используют OPC UA для объединения SCADA-систем с АСУ ТП, что позволяет точнее учитывать потребление энергии и эффективнее распределять ресурсы. В публикации Fine Start подчёркивается значимость OPC UA для повышения производственной эффективности в энергетическом секторе [23]. Например, одна из энергокомпаний использует OPC UA для мониторинга состояния подстанций и генераторов в режиме реального времени, что позволяет оперативно реагировать на изменения и предотвращать аварийные ситуации.

Ещё одним примечательным примером применения OPC UA является транспортная отрасль. Здесь стандарт используется для мониторинга состояния транспортных средств и управления логистическими процессами. Системы управления движением применяют OPC UA для организации обмена данными между транспортными средствами и инфраструктурой, что позволяет повысить координацию движения и уровень безопасности на дорогах. В рамках одного из проектов по внедрению интеллектуальной транспортной системы (ITS) была реализована передача данных между автобусами и центральной системой управления, что обеспечило оптимизацию маршрутов и сокращение времени ожидания для пассажиров.

В сфере здравоохранения OPC UA активно применяется для интеграции медицинского оборудования и систем обработки информации о пациентах. Медицинские устройства способны в реальном времени передавать данные о состоянии пациента в электронные медицинские карты посредством OPC UA, что ускоряет доступ к критически важной информации и способствует улучшению качества обслуживания. В одном из крупных медицинских центров была внедрена система мониторинга, основанная на данном стандарте, позволяющая врачам получать актуальные данные о пациентах прямо на мобильные устройства, что значительно повышает оперативность и эффективность принятия решений.

Стандарт OPC UA продолжает динамично развиваться, отвечая на актуальные технологические вызовы и растущие потребности пользователей.

Перспективные направления его эволюции будут формироваться в соответствии с изменениями в сфере цифровых технологий и автоматизации.

Одной из наиболее значительных тенденций является интеграция с Интернетом вещей (IoT). С увеличением числа подключенных устройств и систем в рамках концепции IoT необходимость в надежном и безопасном обмене данными возрастает. OPC UA предоставляет необходимые механизмы для обеспечения безопасности и совместимости между различными устройствами, что делает его идеальным кандидатом для использования в IoT-экосистемах. Внедрение OPC UA в IoT позволит создать более интеллектуальные системы, способные к самообучению и оптимизации процессов. Например, в умных фабриках устройства могут обмениваться данными через OPC UA, что позволяет оперативно реагировать на изменения в производственном процессе.

Параллельно с этим идет активное развитие облачных технологий. С переходом многих компаний на облачные платформы интеграция OPC UA с облачными решениями становится необходимостью. Это позволит обеспечить более гибкий доступ к данным и возможность анализа больших объемов информации в реальном времени. Облачные решения могут использовать возможности OPC UA для сбора данных с различных источников и их обработки, что значительно улучшит принятие решений на основе данных. Например, компании могут использовать облачные платформы для анализа данных, полученных от различных производственных линий, что позволит оптимизировать процессы и снизить затраты.

Кроме того, кибербезопасность становится критически важным направлением развития. С увеличением числа кибератак на промышленные системы стандарты безопасности OPC UA будут продолжать развиваться. Внедрение новых механизмов защиты данных, таких как многофакторная аутентификация и расширенные методы шифрования, станет важным направлением для повышения уровня безопасности систем на базе OPC UA. Эти меры помогут защитить данные от несанкционированного доступа и обеспечат соответствие современным требованиям кибербезопасности.

Также стоит отметить использование машинного обучения и искусственного интеллекта для повышения уровня безопасности данных. Алгоритмы машинного обучения могут использоваться для анализа трафика и выявления аномалий в поведении пользователей или устройств, что позволяет быстро реагировать на потенциальные угрозы. Это может включать автоматическое обнаружение вторжений или предсказание возможных атак на основе анализа исторических данных.

Развитие стандартов безопасности, связанных с OPC UA, также будет иметь важное значение. В последние годы были разработаны дополнительные спецификации, касающиеся безопасности данных, которые помогают разработчикам интегрировать необходимые меры защиты в свои решения. Это включает в себя рекомендации по безопасной конфигурации серверов и клиентов OPC UA, а также лучшие практики по управлению доступом к данным. Эти стандарты помогают обеспечить единообразие подходов к безопасности среди различных производителей оборудования и программного обеспечения.

Обучение персонала также становится важным аспектом будущего развития стандарта OPC UA. С ростом числа угроз кибербезопасности компании начинают осознавать необходимость обучения своих сотрудников основам безопасности данных. Это включает в себя обучение правильному использованию систем на базе OPC UA, а также осведомленность о потенциальных угрозах и методах их предотвращения. Регулярные тренинги по кибербезопасности помогают повысить уровень осведомленности сотрудников о рисках и способах защиты информации.

OPC серверы находят применения в различных проектах и системах. Особенно OPC серверы снискали популярность в качестве инструмента для формирования баз данных для машинного обучения глубокого обучения, где необходимо обеспечить большой объем информации с низкой частотой измерения параметров.

Исходя из всего вышеперечисленного, было принято решение использовать унифицированную архитектуру OPC UA в качестве основы.

Для реализации OPC UA клиента были рассмотрены следующие языки программирования библиотеки, преимущества и недостатки каждого указанные в таблице 2.

Таблица 2 – Сравнительная таблица языков программирования для реализации OPC UA клиента

Язык программирования	**Библиотеки/Инструменты**	**Критерии выбора**	**Обоснование**
C#	OPC Foundation .NET API	Поддержка Windows, высокая производительность и развитая экосистема	Отлично подходит для разработки под Windows, но ограничен в кроссплатформенности. Хорошо работает с графическими интерфейсами.
Python	opcua, FreeOpcUa	Простота и скорость разработки, поддержка кроссплатформенности, широкий набор библиотек	Python позволяет быстро разрабатывать и тестировать решения, имеет множество библиотек для работы с OPC UA и поддерживает асинхронные операции, что критично для мониторинга данных.

Продолжение таблицы 2

Язык программирования	**Библиотеки/Инструменты**	**Критерии выбора**	**Обоснование**
C++	open62541, OPC UA C++ SDK	Высокая производительность, контроль над ресурсами и сложность разработки	Обеспечивает максимальную производительность, но требует больше времени на разработку и отладку.
Golang	gopcua	Простота синтаксиса, высокая производительность и поддержка параллелизма	Хорошо подходит для высоконагруженных приложений, но имеет меньшую экосистему библиотек по сравнению с Python.
Java	Eclipse Milo	Поддержка кроссплатформенности, многопоточности, а также большая экосистема библиотек	Надежный выбор для корпоративных решений, но может быть менее удобным для быстрого прототипирования.
Delphi	Delphi OPC UA Client	Простота разработки GUI, быстрое создание приложений под Windows	Удобен для разработки под Windows, но ограничен в кроссплатформенности.

Продолжение таблицы 2

Язык программирования	**Библиотеки/Инструменты**	**Критерии выбора**	**Обоснование**
JavaScript/Node.js	node-opcua	Легкость в разработке веб-приложений, поддержка асинхронного программирования и широкое использование в веб-разработке	Отлично подходит для создания веб-клиентов, но может быть менее эффективным для задач с высокой нагрузкой.

Исходя из всего вышеперечисленного, было принято решение использовать язык программирования Python в качестве основы для реализации OPC UA клиента, так как данный язык позволяет быстро создавать прототипы и тестировать функциональность благодаря простоте синтаксиса и богатой стандартной библиотеке. Язык программирования Python является кроссплатформенным, что позволяет запускать клиентские приложения на различных операционных системах это важное преимущество при интеграции с разными устройствами. Благодаря поддержке библиотек, таких как opcua и FreeOpcUa, разработка OPC UA клиента становится значительно проще, позволяя сосредоточиться на логике работы с регистрами ПЛК. Асинхронное программирование, поддерживаемое Python, даёт возможность эффективно обрабатывать множество запросов без блокировки исполнения, что особенно актуально для мониторинга в реальном времени. Развитое сообщество предоставляет доступ к обширной документации и обучающим материалам, упрощая освоение технологий и решение возникающих проблем. Кроме того, Python легко интегрируется с другими языками и системами, выступая универсальным связующим элементом в архитектуре автоматизированных решений.

Во время изучения литературы также были рассмотрены аналоги OPC UA клиентов, такие как: MasterOPC Client, OPC UA.NET Client, IntraSCADA OPC UA Client, Modbus Universal MasterOPC, Multi-Protocol MasterOPC. Результаты сравнения данных приложений представлены в таблице 3.

Таблица 3 – Сравнительная таблица аналогов OPC UA клиентов

Аналог	**Плюсы**	**Минусы**
MasterOPC Client	Поддержка OPC DA, OPC HDA и OPC UA, гибкость в настройке туннелей	Сложность настройки DCOM для удаленных подключений, потенциальные проблемы совместимости с устаревшими системами
OPC UA .NET Client	Кроссплатформенность, безопасность, простота разработки на .NET	Может требовать дополнительных ресурсов для реализации, ограниченная поддержка старых систем
IntraSCADA OPC UA Client	Простота интеграции с SCADA системами, поддержка TCP/IP	Ограниченная гибкость в настройке, возможные проблемы с совместимостью
Modbus Universal MasterOPC	Поддержка ODBC для интеграции с базами данных, масштабирование значений	Неудобная конфигурация, требует остановки рантайма для изменений, нет поддержки OPC UA
Multi-Protocol MasterOPC	Поддержка нескольких протоколов, гибкость в разработке пользовательских протоколов	Сложность реализации, высокие требования к ресурсам, потенциальные затраты на лицензирование

Исходя из результатов сравнения необходимо разработать клиент на основе OPC UA .NET Client с расширением его возможностей для и улучшением гибкости в настройке туннелей. Это обеспечит создание гибкого, безопасного и простого в использовании клиента, который будет удовлетворять потребностям в интеграции с различными промышленными системами.

ВЫВОД К ГЛАВЕ 1

После изучения современной научно-технической, нормативной, методической литературы, затрагивающей исследуемую научно-техническую проблему, было принято решение о разработке OPC UA клиента на основе OPC UA.NET Client с интеграцией SCADA систем на языке программирования Python.

2.АРХИТЕКТУРА И ПРОЕКТИРОВАНИЕ OPC UA

2.1. ЦЕЛИ ПРОЕКТИРОВАНИЯ OPC UA

OPC UA представляет собой независимый от платформ стандарт, позволяющий различным системам и устройствам обмениваться данными посредством передачи сообщений между клиентами и серверами через различные типы сетей. Он поддерживает надёжную и безопасную коммуникацию, обеспечивая идентификацию участников и устойчивость к внешним атакам.

Стандарт определяет наборы сервисов, предоставляемых серверами, при этом каждый конкретный сервер указывает, какие именно сервисы он поддерживает. Передача данных осуществляется с использованием типов, определённых как самим стандартом OPC UA, так и сторонними поставщиками. Серверы создают объектные модели, которые клиенты могут обнаруживать в процессе работы. Кроме текущих данных, сервера могут предоставлять историческую информацию, сигналы тревог и события, уведомляя клиентов о важных изменениях в системе.

OPC UA поддерживает подключение к различным коммуникационным протоколам и допускает передачу данных в различных кодировках – от бинарных структур до XML-документов. Это обеспечивает мобильность и эффективность обмена информацией. Формат данных может задаваться как самим стандартом, так и другими организациями или поставщиками. Через адресное пространство клиенты могут запрашивать метаданные, описывающие формат передаваемой информации, что особенно полезно в случаях, когда клиенту не требуется заранее знать структуру данных – она может быть определена во время выполнения.

Стандарт предоставляет единую интегрированную адресную область и модель обслуживания, объединяя доступ к данным, тревогам, событиям и истории через унифицированный набор сервисов с встроенной моделью безопасности. OPC UA также поддерживает множество взаимосвязей между узлами, поз-

воляя формировать несколько иерархий представления данных, адаптированных под разные группы клиентов. Такая гибкость, наряду с возможностью описания типов объектов и информационных моделей, делает стандарт применимым к широкому спектру задач. Как показано на рисунке 1, OPC UA используется не только на уровне взаимодействия SCADA, PLC и DCS, но и для расширения функциональной совместимости на более высоких уровнях автоматизации.

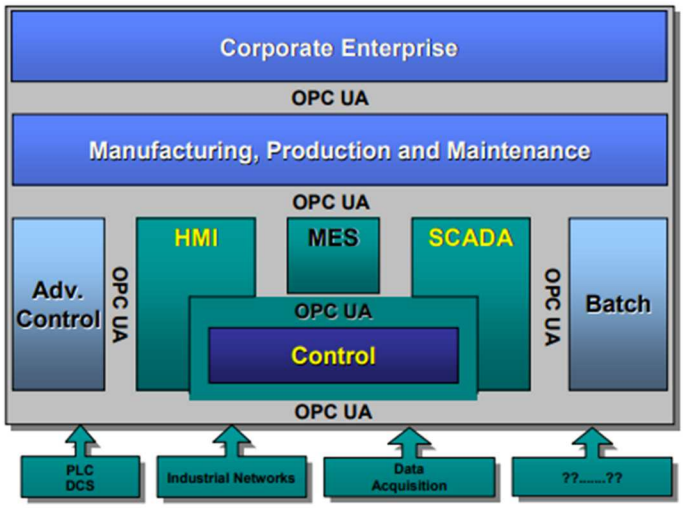

Рисунок 1 – Целевые приложения OPC UA

OPC UA разработан для обеспечения надежности публикуемых данных. Важной особенностью всех серверов OPC является возможность публикации данных и уведомлений о событиях. OPC UA предоставляет клиентам механизмы быстрого обнаружения и восстановления после сбоев связи, связанных с этими передачами, без необходимости ждать длительных перерывов в работе, предусмотренных базовыми протоколами.

OPC UA предназначен для поддержки широкого спектра серверов, от ПЛК на заводе до корпоративных серверов. Эти серверы отличаются широким диапазоном размеров, производительности, платформ выполнения и функциональных возможностей. Таким образом, OPC UA определяет полный набор возможностей, и серверы могут реализовывать часть этих возможностей.

Для повышения функциональной совместимости OPC UA определяет подмножества, называемые профилями, которым серверы могут предъявлять требования о соответствии. Затем клиенты могут находить профили сервера и адаптировать свои взаимодействия с этим сервером на основе профилей

Спецификации OPC UA имеют многоуровневую структуру, чтобы отделить базовый дизайн от базовой вычислительной технологии и сетевого транспорта. Это позволяет при необходимости сопоставлять OPC UA с будущими технологиями, не нарушая при этом базовый дизайн.

Существуют две кодировки данных и три транспортных протокола:

- XML;
- двоичный код UA;
- OPC UA TCP;
- SOAP/HTTP;
- HTTPS.

Клиенты и серверы, поддерживающие несколько транспортов и кодировок, позволят конечным пользователям принимать решения о компромиссах между производительностью и совместимостью веб-служб XML во время развертывания вместо того, чтобы эти компромиссы определялись поставщиком OPC во время определения продукта.

OPC UA разработан как средство миграции для клиентов и серверов OPC, работающих на базе Microsoft COM. При разработке OPC-UA были приняты меры для того, чтобы существующие данные, предоставляемые OPC COM-серверы (DA, HDA и A&E) могут быть легко подключены и доступны через OPC UA. Поставщики могут решить перенести свои продукты на OPC UA или использовать внешние оболочки для преобразования из OPC UA в OPC UA, OPC COM в OPC UA и наоборот. Каждая из предыдущих спецификаций OPC определяла свою собственную модель адресного пространства и свой собственный набор сервисов. OPC UA объединяет предыдущие модели в единое интегрированное адресное пространство с единым набором сервисов.

2.2. АРХИТЕКТУРА OPC UA СИСТЕМЫ

Архитектура систем OPC UA моделирует клиенты и серверы OPC UA как взаимодействующих партнеров. Каждая система может содержать несколько клиентов и серверов. Каждый клиент может одновременно взаимодействовать с одним или несколькими серверами, а каждый сервер может одновременно взаимодействовать с одним или несколькими клиентами. Приложение может объединять серверные и клиентские компоненты, чтобы обеспечить взаимодействие с другими серверами и клиентами. На рисунке 4 показана архитектура, включающая в себя комбинированный сервер и клиент.

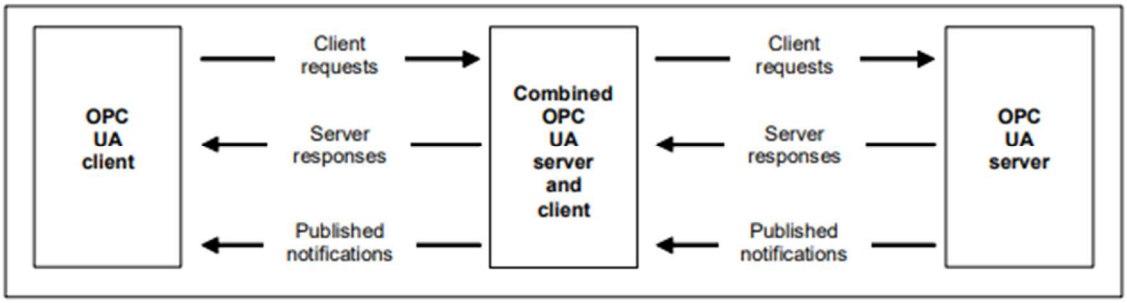

Рисунок 4 – Архитектура OPC UA системы

2.3. АРХИТЕКТУРА OPC UA КЛИЕНТА

Клиентская архитектура OPC UA моделирует конечную точку взаимодействия клиент-сервер. На рисунке 5 показаны основные элементы OPC UA клиента и то, как они связаны друг с другом.

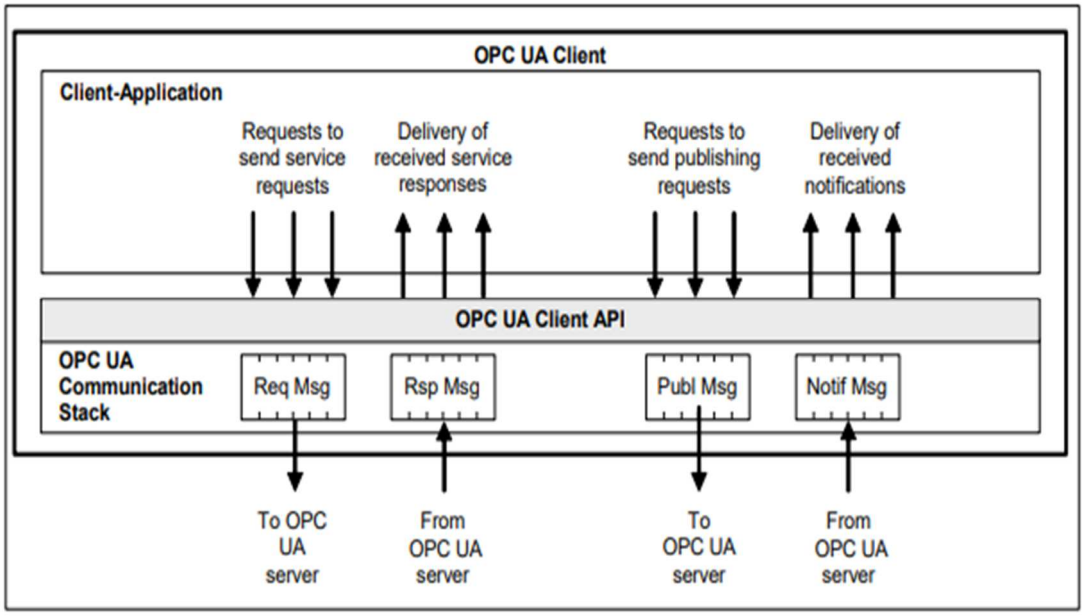

Рисунок 5 – Архитектура OPC UA клиента

Клиентское приложение – это код, который реализует функции клиента. Оно использует API OPC UA клиента для отправки и получения запросов на обслуживание OPC UA и ответов на сервер OPC UA.

API OPC UA клиента – это внутренний интерфейс, который изолирует код клиентского приложения от коммуникационного стека OPC UA. Коммуникационный стек OPC UA преобразует клиент API OPC UA вызывает сообщения и отправляет их через базовый коммуникационный объект на сервер по запросу клиентского приложения. Коммуникационный стек OPC UA также получает ответные сообщения и уведомления от базового объекта связи и передает их клиентскому приложению через клиентский API OPC UA.

В роли клиента для сервера OPC UA выступает программа, которая запрашивает у него информацию. Примером такой программы может служить система SCADA. Запросы к серверу OPC отправляются через внутренний интерфейс, который служит изолирующей прослойкой между программой-клиентом и коммутационным стеком. Коммутационный стек преобразует запросы клиента в сообщения, необходимые для вызова соответствующих сервисов на сервере. После получения ответа от сервера коммутационный стек передает результат обратно в программу-клиент.

Архитектура OPC UA также позволяет осуществлять обмен данными между двумя серверами. Для этого один из них становится клиентом, а другой – сервером. Это позволяет объединить несколько серверов в цепочку, где каждый элемент может одновременно выполнять роль как клиента, так и сервера с разных сторон цепочки.

2.4. АРХИТЕКТУРА OPC UA СЕРВЕРА

Архитектура сервера OPC UA моделирует конечную точку взаимодействия клиент-сервер. Рисунок 6 [17] иллюстрирует основные элементы сервера OPC UA и то, как они связаны друг с другом.

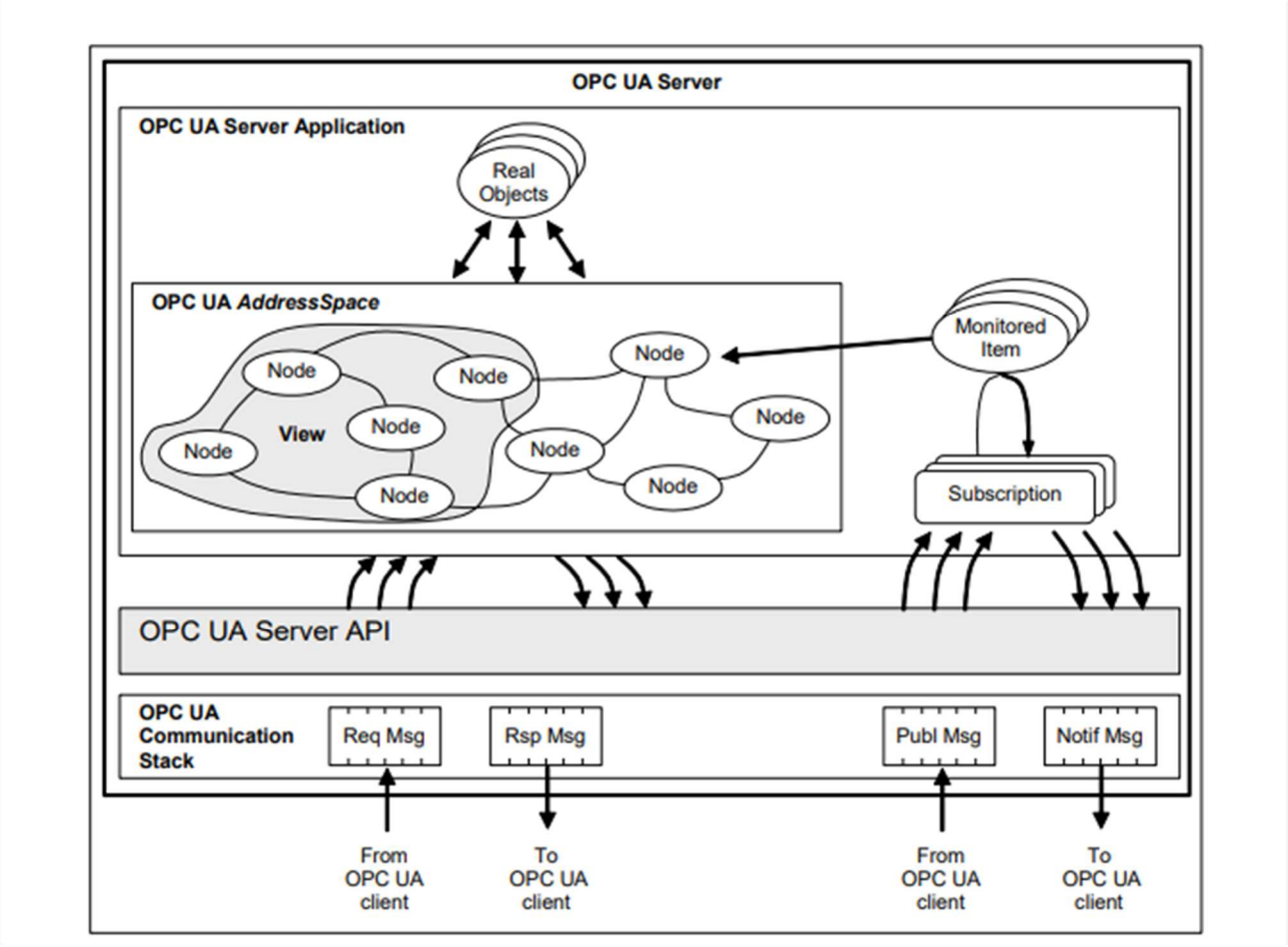

Рисунок 6 – Архитектура OPC UA сервера

На данном рисунке реальными объектами выступают контроллеры (ПЛК), устройства ввода/вывода и другие устройства, способные передавать данные через сервер OPC. Серверное приложение представляет собой программную

реализацию функций, которые должен выполнять сервер. Взаимодействие между клиентом и сервером OPC UA происходит через интерфейс прикладной программы посредством отправки запросов и получения ответов.

Обмен данными между сервером и клиентом может осуществляться двумя основными способами:

Ответ на мгновенные запросы: Клиент отправляет запрос, а сервер предоставляет ответ немедленно.

Схема «издатель – подписчик»: В этом режиме клиентская программа подписывается на получение определенных данных. Сервер обязан предоставлять эти данные по мере их обновления или изменения значений.

Для реализации схемы «издатель – подписчик» клиентская программа устанавливает «подписку» на определенные данные. Сервер мониторит соответствующие узлы и связанные с ними реальные объекты для обнаружения изменений в данных, событиях или аварийных сигналах. Как только обнаруживаются изменения, сервер генерирует уведомление, которое передается клиенту по каналу подписки.

2.5. АДРЕСНОЕ ПРОСТРАНСТВО OPC UA

Адресное пространство в OPC UA моделируется как совокупность узлов, доступных клиентам через сервисы стандарта, включающие интерфейсы и методы. Узлы используются для представления реальных объектов, их типов и взаимосвязей между ними.

Серверы имеют возможность свободно организовывать структуру узлов в адресном пространстве, исходя из собственных требований. С помощью ссылок между узлами они могут формировать как иерархические структуры, так и более сложные сетевые топологии. В некоторых профилях допускается неполная реализация всех узлов общего доступа.

Частью адресного пространства являются представления (views) – подмножества узлов, которые сервер делает доступными для клиента. Это позволяет ограничить объём данных, обрабатываемых в запросах, и упростить навигацию.

По умолчанию клиент получает доступ ко всему адресному пространству, однако сервер может определить дополнительные представления, скрывающие определённые узлы или связи. Клиенты могут просматривать доступные представления и изучать их структуру, как правило, представленную в виде дерева, что облегчает ориентацию в модели данных.

Адресное пространство OPC UA поддерживает реализацию информационных моделей, что достигается за счёт следующих механизмов:

1. Использование ссылок между узлами, позволяющих устанавливать связи между объектами в адресном пространстве.
2. Применение узлов типа ObjectType для предоставления семантической информации, описывающей реальные объекты.
3. Возможность подклассификации типов с помощью иерархий узлов ObjectType.
4. Определения типов данных, доступные в адресной области, что позволяет использовать отраслевые стандарты типов.
5. Сопутствующие стандарты OPC UA, предоставляющие отраслевым организациям возможность задавать правила представления собственных моделей данных в адресном пространстве серверов.

Отслеживаемые объекты – это элементы, создаваемые клиентом на сервере для слежения за определёнными узлами и соответствующими им объектами в реальном мире. При изменении данных или возникновении событий такие объекты формируют уведомления, которые передаются клиенту по механизму подписки.

Конечная точка на сервере, откуда публикуются уведомления, называется подпиской. Клиенты управляют частотой этих уведомлений, отправляя сообщения о публикации.

2.6. АНАЛИЗ ПРОЕКТИРУЕМОЙ СИСТЕМЫ

В ходе разработки OPC UA клиента для считывания данных из регистров программируемого логического контроллера были сформулированы основные

функциональные и нефункциональные требования. Они обеспечивают основу для создания надёжного и эффективного клиента, взаимодействующего с серверами OPC UA.

2.6.1. ФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Функциональные требования охватывают несколько ключевых направлений. В первую очередь, клиент должен обеспечивать подключение к OPC UA серверу по заданному URL с поддержкой аутентификации – как по логину и паролю, так и с использованием цифровых сертификатов для повышения уровня безопасности.

Клиент должен поддерживать считывание значений регистров с сервера, обрабатывая различные типы данных, включая целочисленные, вещественные, строковые и логические. Также должна быть реализована возможность записи данных в регистры с аналогичной поддержкой типов.

Для обеспечения своевременного реагирования на изменения клиент должен включать механизм подписки, позволяющий получать уведомления в реальном времени. Также важно предусмотреть настройку интервалов опроса для периодического получения информации.

Обработка ошибок – критически важный элемент. Клиент должен надёжно обрабатывать сбои соединения и выводить пользователю соответствующие уведомления. Логирование ошибок и событий необходимо для анализа и последующей оптимизации.

Интерфейс должен быть интуитивно понятным и предоставлять возможность настройки параметров подключения без изменения исходного кода. Визуализация данных в формате таблиц и графиков также должна быть предусмотрена.

Поддержка многопоточности необходима для одновременного выполнения операций чтения, записи и обработки пользовательских запросов. Кроме того, клиент должен уметь сохранять полученные данные в локальную базу или файл для последующего анализа и формирования отчётов.

Наконец, клиент обязан быть совместимым с различными версиями OPC UA серверов и поддерживать ключевые функции протокола, включая механизм публикации/подписки (PubSub).

2.6.2. НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

Нефункциональные требования ориентированы на обеспечение качества и надёжности работы клиента. Одним из ключевых аспектов является производительность: клиент должен обрабатывать операции чтения и записи не более чем за 2 секунды, даже при высоком уровне пользовательской нагрузки. Также важно обеспечить одновременную обработку нескольких запросов без потери эффективности.

Надёжность системы должна гарантировать не менее 99,9% времени безотказной работы, обеспечивая стабильный доступ к данным. В случае разрыва соединения должен автоматически выполняться его восстановление.

В области безопасности предусматривается использование защищённых каналов передачи с 256-битным шифрованием, а также аутентификация пользователей и устройств посредством цифровых сертификатов.

Интерфейс клиента должен быть интуитивно понятным, с возможностью настройки параметров подключения без необходимости редактировать код. Также важно наличие встроенной справки и подсказок для пользователей.

Клиент должен быть масштабируемым, что предполагает стабильную работу при росте объёмов данных и количества подключений, а также возможность расширения функциональности без серьёзной переработки архитектуры.

Кроссплатформенность – ещё одно важное требование: клиент должен корректно работать на операционных системах Windows, Linux и macOS.

Система логирования должна фиксировать действия пользователя, ошибки и события для последующего анализа. Необходим также механизм мониторинга состояния соединения и получаемых данных в реальном времени.

Все функции клиента должны сопровождаться подробной документацией, включая описание API, инструкции по установке и настройке, а также примеры использования.

2.7. ПРОЕКТИРОВАНИЕ

На этапе проектирования OPC UA клиента для считывания данных из регистров программируемого логического контроллера были выбраны подходы и шаблоны проектирования, обеспечивающие масштабируемость, устойчивость и удобство разработки. Ключевое внимание было уделено разделению логики на уровни, что значительно упрощает поддержку и последующее расширение функциональности.

Для реализации использован объектно-ориентированный подход, при котором каждый модуль оформляется в виде набора классов и сервисов. Это позволяет изолировать бизнес-логику и облегчает внесение изменений. Архитектура клиента построена по сервисной модели, что соответствует философии OPC UA и способствует созданию гибкого, масштабируемого решения. Клиент будет совместим с различными OPC UA серверами и способен взаимодействовать с широким спектром устройств и контроллеров.

Архитектура клиента включает в себя следующие основные компоненты:

- модуль подключения: отвечает за установление и поддержание соединения с OPC UA сервером;
- модуль считывания данных: реализует функции для получения значений регистров с сервера;
- модуль записи данных: позволяет записывать значения в регистры на сервере;
- модуль обработки ошибок: управляет исключениями и уведомляет пользователя о возникших проблемах;
- интерфейс пользователя: графический интерфейс для взаимодействия с пользователем, настройки параметров и отображения данных.

Паттерны проектирования:

- паттерн «Адаптер» используется для унификации взаимодействия с различными типами OPC UA серверов, позволяя скрыть их особенности и упростить разработку клиента. Это важно при поддержке серверов разных классов, от встроенных до стандартных;
- паттерн «Фабрика» обеспечивает гибкое создание клиентов для различных конфигураций и типов подключений. Он упрощает расширение системы без изменения существующего кода, повышая масштабируемость;
- паттерн «Наблюдатель» реализует подписку на изменения данных, что позволяет клиенту получать обновления в реальном времени и оперативно реагировать на события – критически важно в системах промышленной автоматизации.

Для взаимодействия с OPC UA сервером используется библиотека opcua, обеспечивающая надёжное и безопасное соединение. Для логирования и обработки ошибок применяется модуль logging, что позволяет отслеживать работу клиента и эффективно диагностировать возможные неисправности. Библиотека opcua поддерживает сервис-ориентированную архитектуру, соответствующую стандарту OPC UA, что упрощает интеграцию с различными устройствами и системами. Кроме того, она предоставляет встроенные механизмы безопасности, включая шифрование и аутентификацию.

Каждый функциональный компонент, такой как подключение к серверу, чтение или запись данных, реализуется в виде отдельного модуля с соответствующими классами и сервисами. Такой подход обеспечивает чёткое разделение бизнес-логики и упрощает поддержку системы. Благодаря модульной архитектуре можно добавлять новые функции без значительных изменений в существующем коде, что особенно важно для адаптации клиента к различным задачам и конфигурациям.

Для защиты данных используется шифрование и аутентификация на основе цифровых сертификатов, что предотвращает несанкционированный доступ и обеспечивает стабильность соединения. Стандарт OPC UA предусматривает

различные уровни безопасности, включая подписание и шифрование сообщений, что позволяет обеспечить целостность и конфиденциальность передаваемой информации. Внедрение этих механизмов в клиенте соответствует современным требованиям к промышленной безопасности.

В качестве направлений для развития рассматривается внедрение инструментов визуализации данных, добавление системы автоматических уведомлений при изменении значений регистров, а также интеграция с популярными аналитическими платформами для расширения возможностей отчётности. Также планируется реализация поддержки дополнительных протоколов и интерфейсов для взаимодействия с другими системами и устройствами.

Подводя итог, можно отметить, что разработанный клиент отличается модульной архитектурой, удобством использования и возможностью масштабирования, что делает его универсальным решением для задач промышленной автоматизации. Он обеспечивает надёжное и безопасное взаимодействие с OPC UA серверами, способствуя повышению эффективности управления технологическими процессами и оптимизации производственных систем.

ВЫВОД К ГЛАВЕ 2

Была спроектирована архитектура клиента, а также проведен анализ проектируемой системы. Были определены функциональные и нефункциональные требования к клиенту, набор паттернов, используемых в разработке и предполагаемые библиотеки.

3. РЕАЛИЗАЦИЯ

3.1. СТЕК ИСПОЛЬЗУЕМЫХ ТЕХНОЛОГИЙ

В качестве языка программирования для разработки клиента был выбран Python. Как упоминалось ранее, программы на этом языке не требуют предварительной компиляции и одинаково исполняются на любых операционных системах, для которых доступен интерпретатор. Python обладает широким набором библиотек, существенно упрощающих разработку, а сам интерпретатор написан на языке C, что позволяет скомпилировать его под любую ОС.

На официальном сайте доступны версии интерпретатора для Windows, macOS и наиболее распространённых дистрибутивов Linux.

Для работы с базой данных используется сервер PostgreSQL, который, подобно Python, является кроссплатформенным решением. Он разработан для тех же операционных систем, обеспечивая универсальность и гибкость при развертывании приложения.

На рисунке 7 представлена структурная схема опроса устройств через систему ввода/вывода.

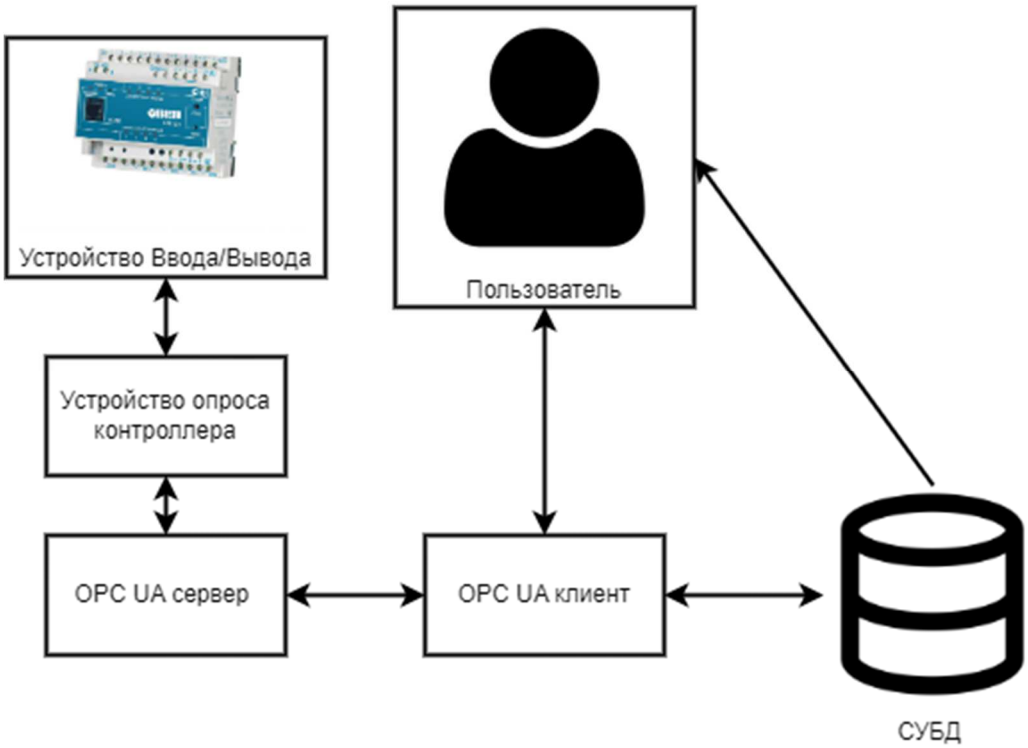

Рисунок 7 – Структурная схема

Основное применение данного клиента можно описать в виде структурной схемы, представленной на рисунке 8.

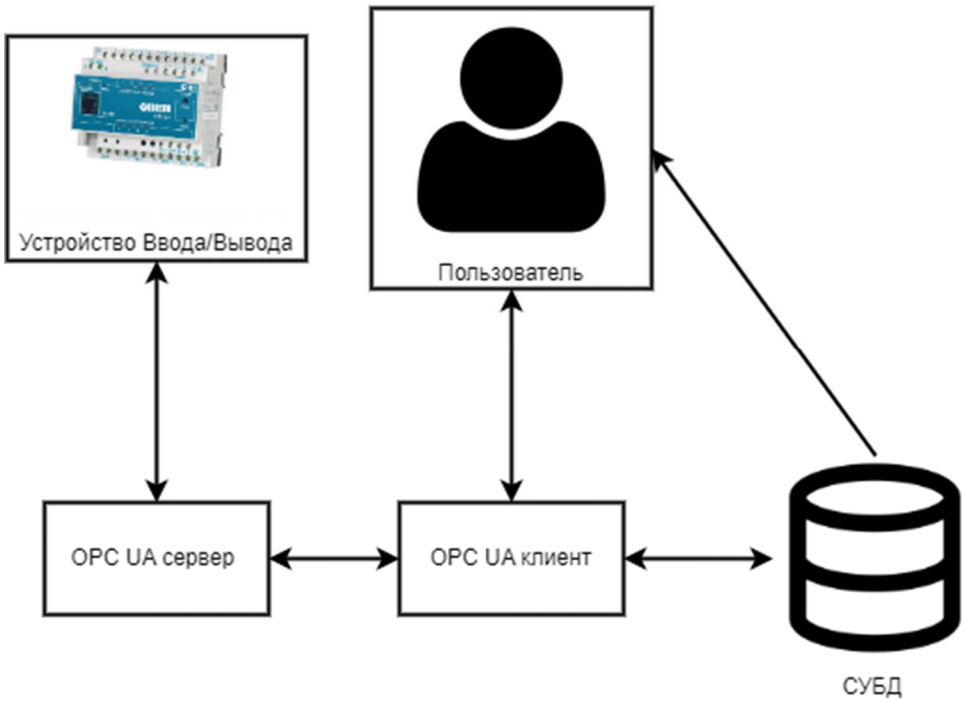

Рисунок 8 – Структурная схема

3.2. РАЗРАБОТКА БАЗЫ ДАННЫХ

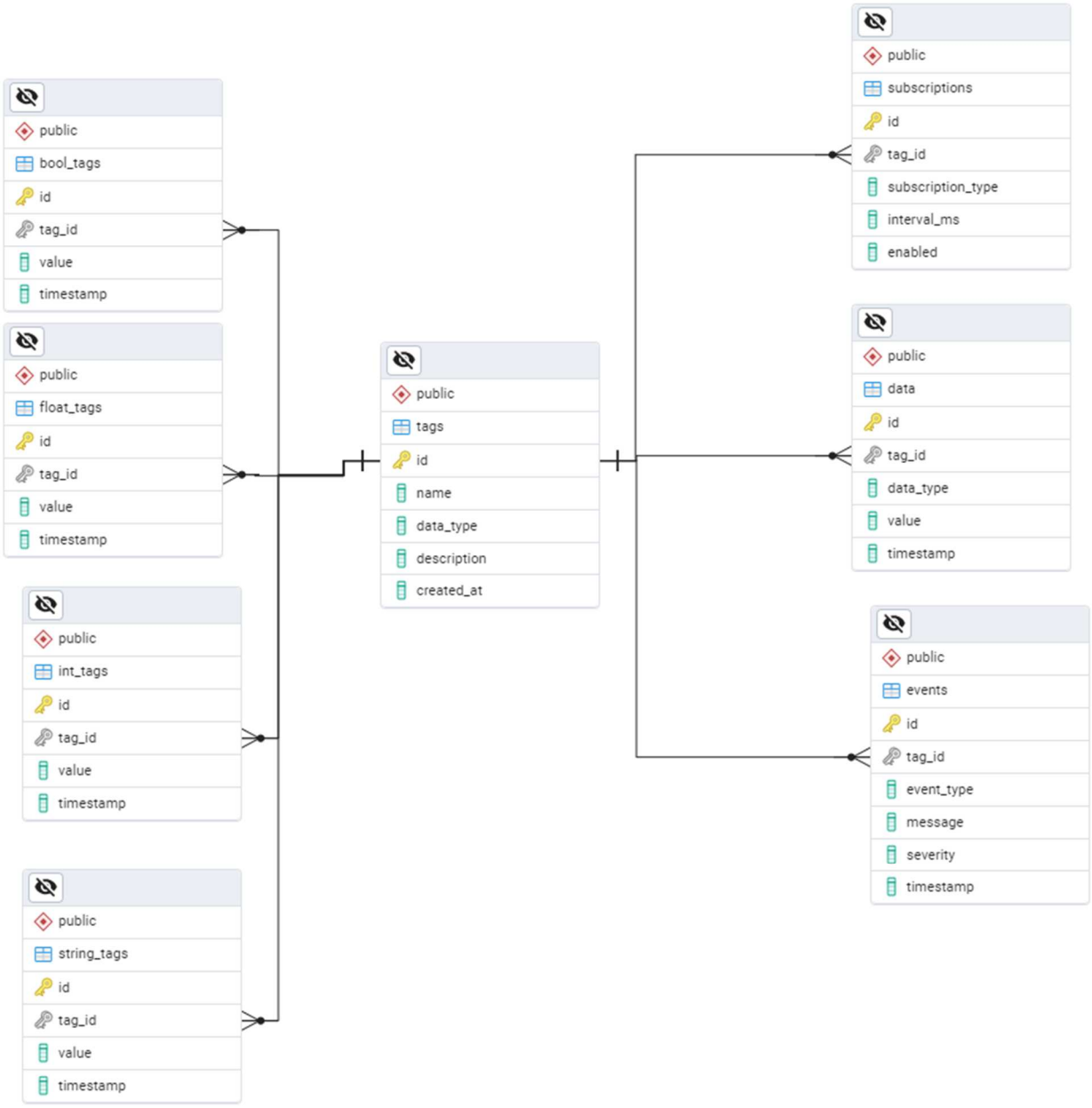

Рисунок 9 – ERD диаграмма базы данных

Выбранная структура базы данных обусловлена особенностями работы сервера OPC UA, который поддерживает множество типов данных. Использование только одного типа, например string, было бы неэффективным из-за большого объёма памяти, занимаемого такими значениями. Кроме того, в различных узлах могут присутствовать теги с одинаковыми именами, что усложняет последующий анализ полученной информации.

В данной структуре реализовано несколько таблиц: таблица событий для фиксации действий клиента и ПЛК, таблица подписок для управления клиентскими подписками, а также таблица исторических данных OPC UA сервера. Запись в конкретную таблицу осуществляется в зависимости от типа данных на сервере: значения типа integer, float, bool записываются в соответствующие таблицы, тогда как строки и прочие данные направляются в таблицу string_tags.

Сущность tags содержит основную информацию о каждом уникальном теге и включает следующие атрибуты:

- id является первичным ключом и показывает порядковый номер тега, хранимого в данной таблице;
- name является названием тега, значение которого записывается
- data_type хранит тип данных записанного тега;
- description является описанием переменной, установленное сервером;
- created_at дата и время записи значения. Необходимо для дальнейшего анализа истории изменения записываемого значения.

Сущности bool_tags, float_tags, int_tags, string_tags имеют одинаковые атрибуты:

- id – первичный ключ, который указывает на номер записанного значения;
- tag_id ссылка на таблицу tags с номером данного тега;
- value – непосредственно значение тэга, которое хранится в базе данных;

В зависимости от сущности (float, int, bool или string) имеет свой тип данных.

- timestamp – дата и время записи значения. Необходимо для дальнейшего анализа истории изменения записываемого значения.

Сущность subscriptions содержит информацию об управлении подписками на данные от OPC UA сервера. Атрибуты данной сущности:

- id первичный ключ, указывающий порядковый номер подписки, хранимый в этой таблице;
- tag_id ссылка на таблицу tags для указания тега, на который подписаны;
- subscription_type тип подписки для спонтанных обновлений или для периодического опроса;
- interval_ms интервал опроса в миллисекундах;
- enabled флаг указывающий, включена ли подписка. По умолчанию подписка включена.

Сущность events хранит информацию о событиях, связанных с работой ПЛК или OPC UA клиента. Атрибуты данной сущности:

- id первичный ключ, указывающий на порядковый номер события;
- tags_id ссылка на таблицу tags для указания тега, с которым связано событие;
- event_type тип события;
- message сообщение, выдаваемое при событии;
- severity – уровень серьезности события;
- timestamp дата и время записи о возникновении события. Необходимо для дальнейшего анализа истории изменения записываемого значения.

Сущность data хранит записи о исторических данных, полученных от OPC UA сервера. Атрибуты сущности:

- id первичный ключ, указывающий на порядковый номер записи;
- tag_id ссылка на таблицу tags для указания тега, к которому относиться запись данных;
- value содержит значение о записи в текстовом формате для всех типов;
- timestamp дата и время записи о возникновении записи данных.

3.3. ОПИСАНИЕ ИСПОЛЬЗУЕМЫХ БИБЛИОТЕК

Для работы программы необходимо наличие следующих библиотек:

- time;
- psycopg2;
- opcua;
- tkinter;
- stdiomask;
- matpolib.

Библиотека time [12] позволяет работать со временем.

Библиотека psycopg2 [4] позволяет осуществлять работу с PostgreSQL сервером. Данный модуль один из самых популярных и стабильных модулей для работы с PostgreSQL, а также он используется в большинстве фреймворков Python и Postgres, потокобезопасен и спроектирован для работы в многопоточных приложениях. С помощью неё возможно осуществлять подключение к базам данных и выполнение SQL запросов с отображением полученных результатов.

Библиотека opcua [10] позволяет подключаться к OPC UA серверу и получать с него данные. Данная библиотека включает в себя как классы для разработки сервера OPC UA, так и классы разработки OPC UA клиента. В разработанном программном обеспечении используется только классы клиента OPC UA.

Стандартная библиотека tkinter [13] используется для создания графического интерфейса программы. С её помощью выводится структура сервера, данные отслеживаемых тегов, а также данные о подписках и событиях.

Библиотека stdiomask [11] необходима для ввода паролей к терминалу stdio и отображения маски с целью недопущения его распространения.

Библиотека matpolib служит для вывода на экран графика изменения значений выбранного тега.

3.4. ОПИСАНИЕ РЕАЛИЗАЦИИ И АЛГОРИТМА РАБОТЫ ПРОГРАММЫ

Перед запуском клиента необходимо выполнить следующие действия:

1. Установить интерпретатор Python.
2. При отсутствии – установить и настроить сервер PostgreSQL для хранения данных.
3. С помощью менеджера пакетов pip, входящего в состав Python, установить все необходимые библиотеки. Установка выполняется через командную строку или терминал в зависимости от операционной системы с использованием команды: pip install «название_библиотеки».

Сначала происходит подключение к Базе данных PostgreSQL. Код программы указан в листинге А.1 приложения А. Данная часть программы отвечает за подключение к базе данных через метод _connect используя параметры конфигурации для выполнения операция и взаимодействия с ней. Методы _create_tables и _check_database_structure создают таблицу или проверяют их структуру на корректность, в противном случае таблица пересоздается. Программа предоставляет методы для создания, обновления и получении информации о тегах, логирования событий, получения истории операций с узлами и управления подписками. Класс включает метод close для безопасного закрытия соединения с базой данных.

Далее реализована программа основных функций OPC UA клиента. Исходный код программы указан в листинге Б.1 приложения Б. Даная часть программы является основной. В данном коде содержатся функции для подключения к серверу OPC UA с аунтификацией, валидацией, парсингом и проверкой доступности сервера в методе connect. Реализованы функции чтения и записи значений узлов обработкой ошибок и повторными попытками в read_node и write_node с замером времени выполнения. Метод subscribe реализует безопасную подписку на узел с проверкой поддержки и вызова соответствующих функций обратного

вызова при изменении данных. Также в клиенте есть возможность обработки событий, таких как изменения данных, в отдельном потоке и просмотр доступных узлов на сервере.

После указанных выше действий был реализован графический интерфейс приложения, в котором происходит вся работа пользователя. Исходный код программы, запускающий графический интерфейс, представлен в листинге В.1 приложения В. Пользовательский интерфейс представлен на рисунке 10.

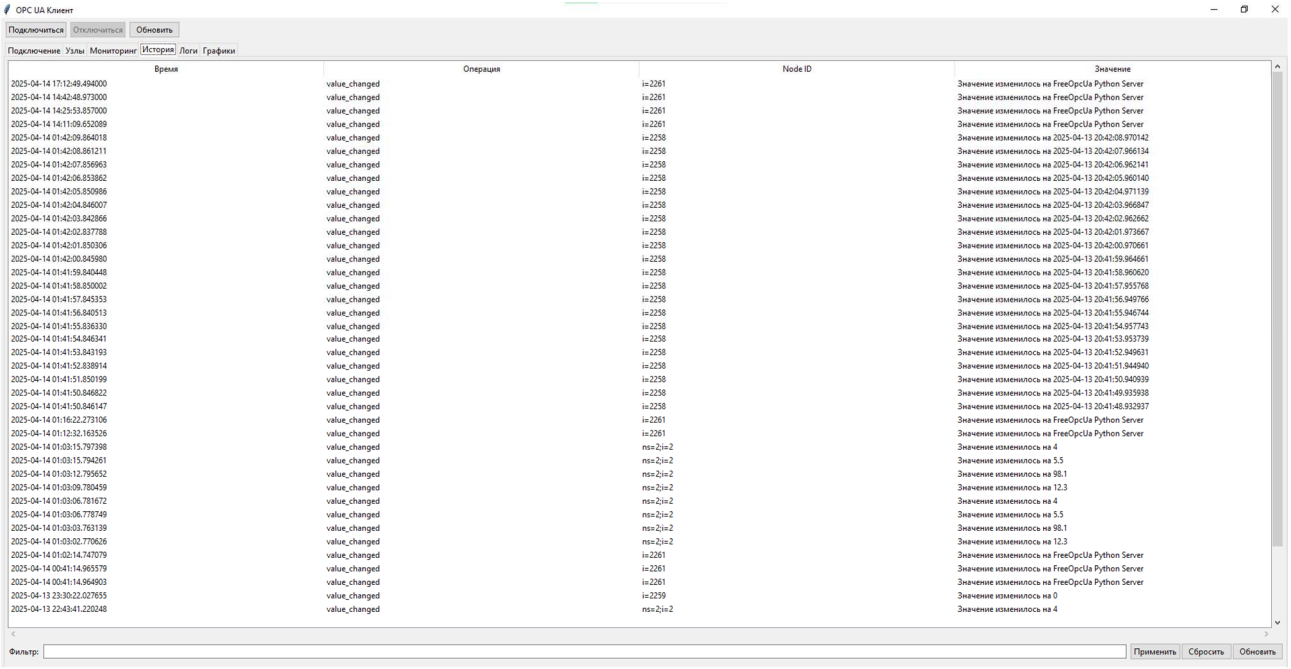

Рисунок 10 – Пользовательский интерфейс

После завершения работы программы выполняется безопасное отключение от OPC UA сервера и БД PostgreSQL с помощью следующих команд:

- cnx.close();
- client.disconnect().

В пользовательском интерфейсе реализована экранная форма подключения к OPC UA серверу с конченой точкой сервера по умолчанию и вводом логина, пароля. Экранная форма подключения представлено на рисунке 11.

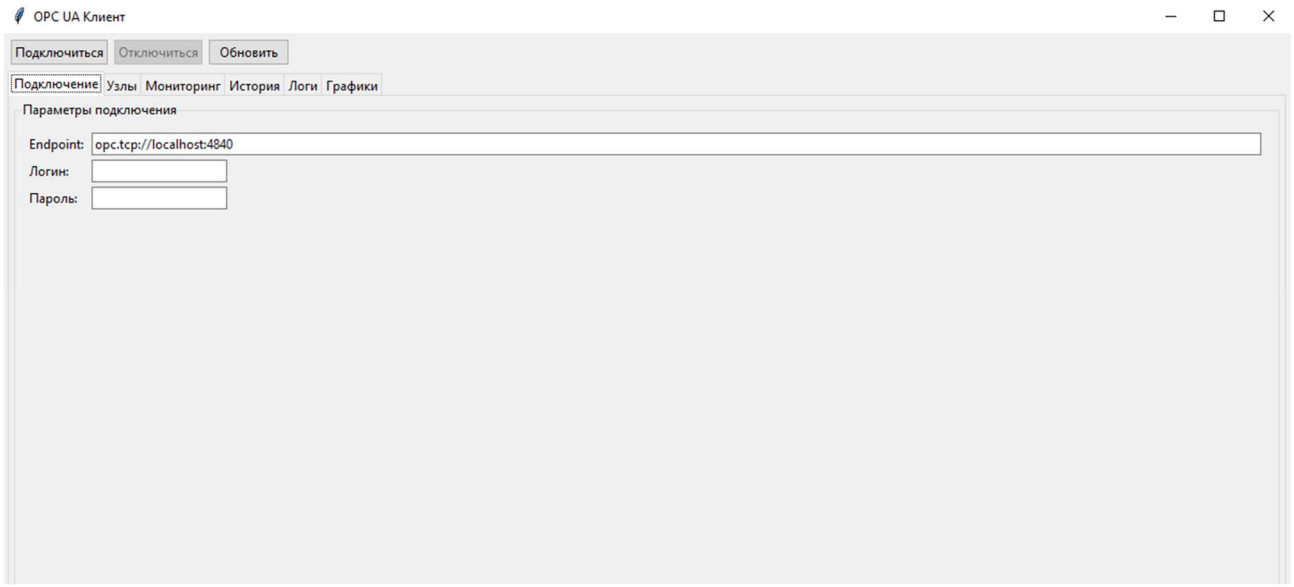

Рисунок 11 – Экранная форма подключения к серверу

Также в интерфейсе реализовано экранная форма управления узлами для чтения и записи узлов, с подробной информацией о теге и подписки на них. Экранная форма управления узлами представлено на рисунке 12.

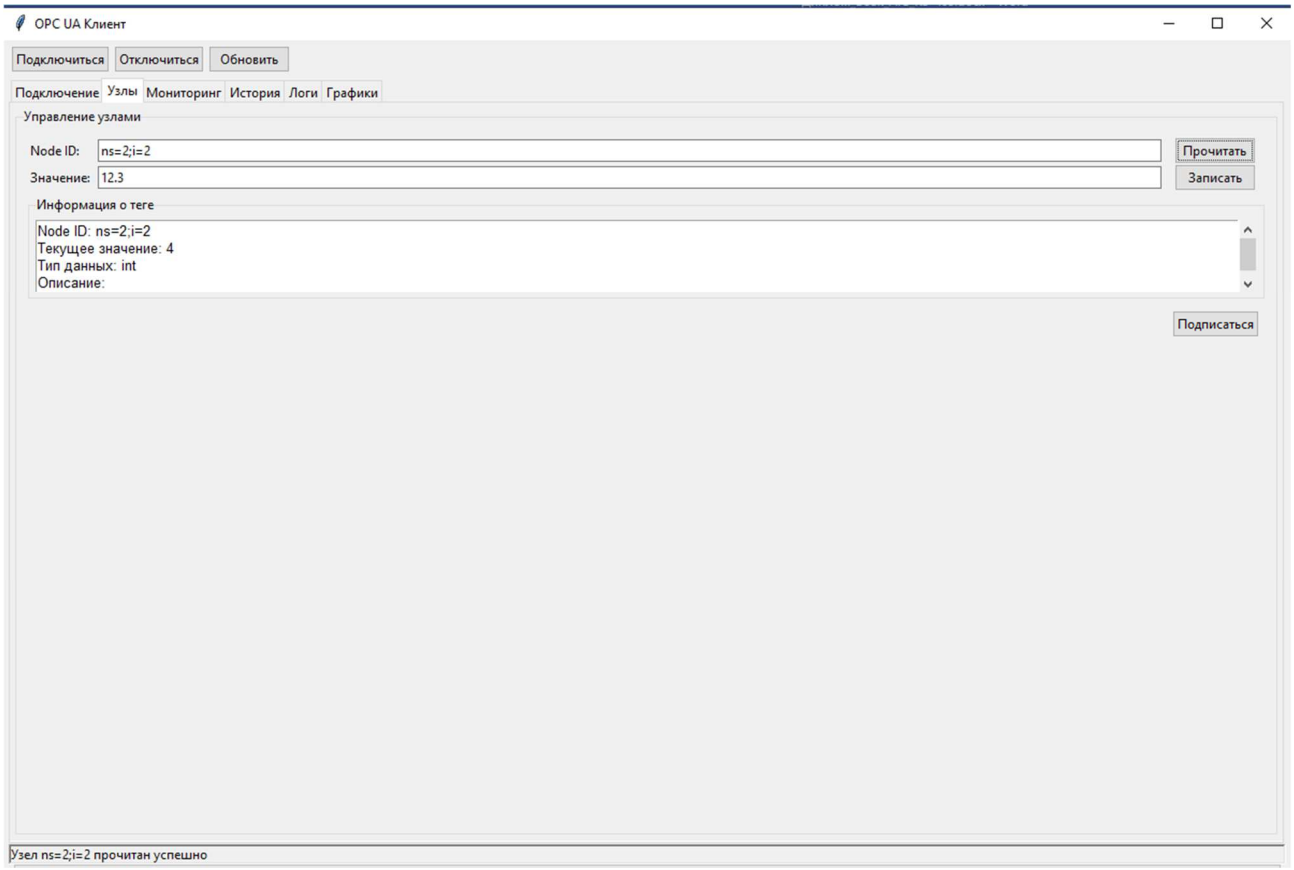

Рисунок 12 – Экранная форма управления узлами

Во вкладке «Мониторинг» реализовано управление активными подписками и мониторинг событий сервера и ПЛК. Вкладка «Мониторинг» представлена на рисунке 13.

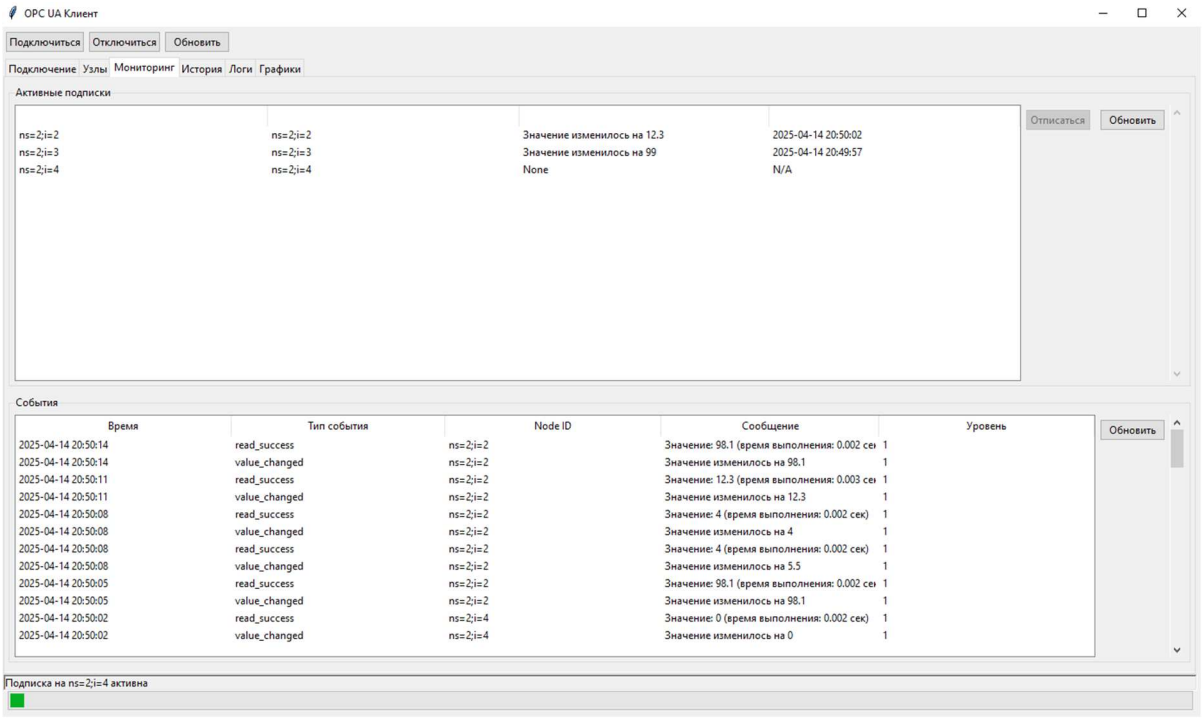

Рисунок 13 – Экранная форма управления подписками и событиями

Во вкладке «История» реализован мониторинг всех операций, проведённых с узлами. Вкладка «История» представлена на рисунке 14.

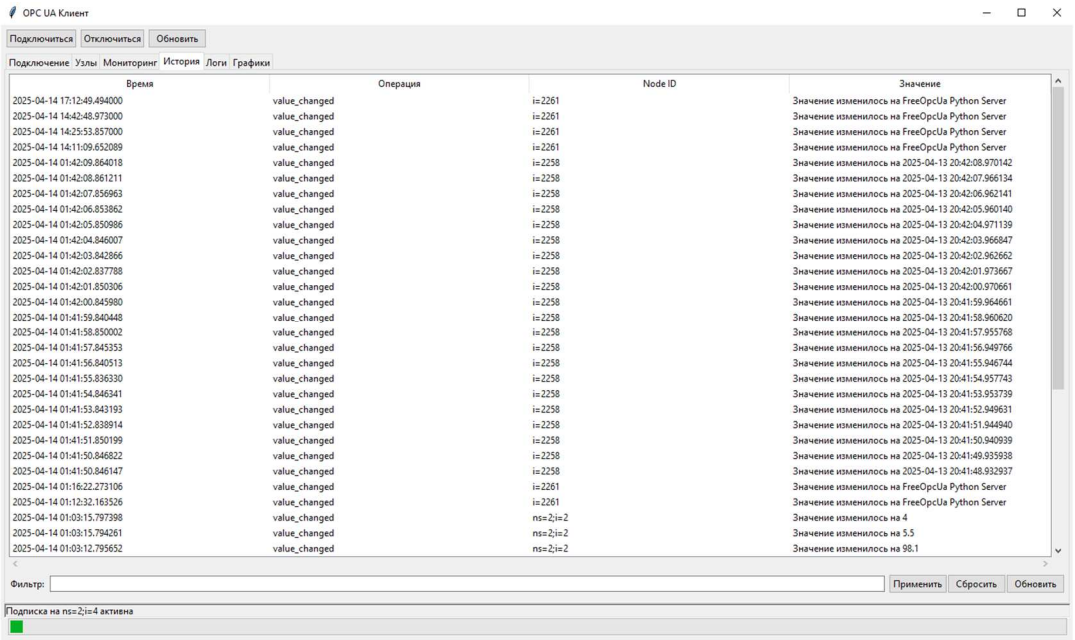

Рисунок 14 – Вкладка «История»

Во вкладке «Логи» реализована запись логов клиента. Вкладка «Логи» представлена на рисунке 15.

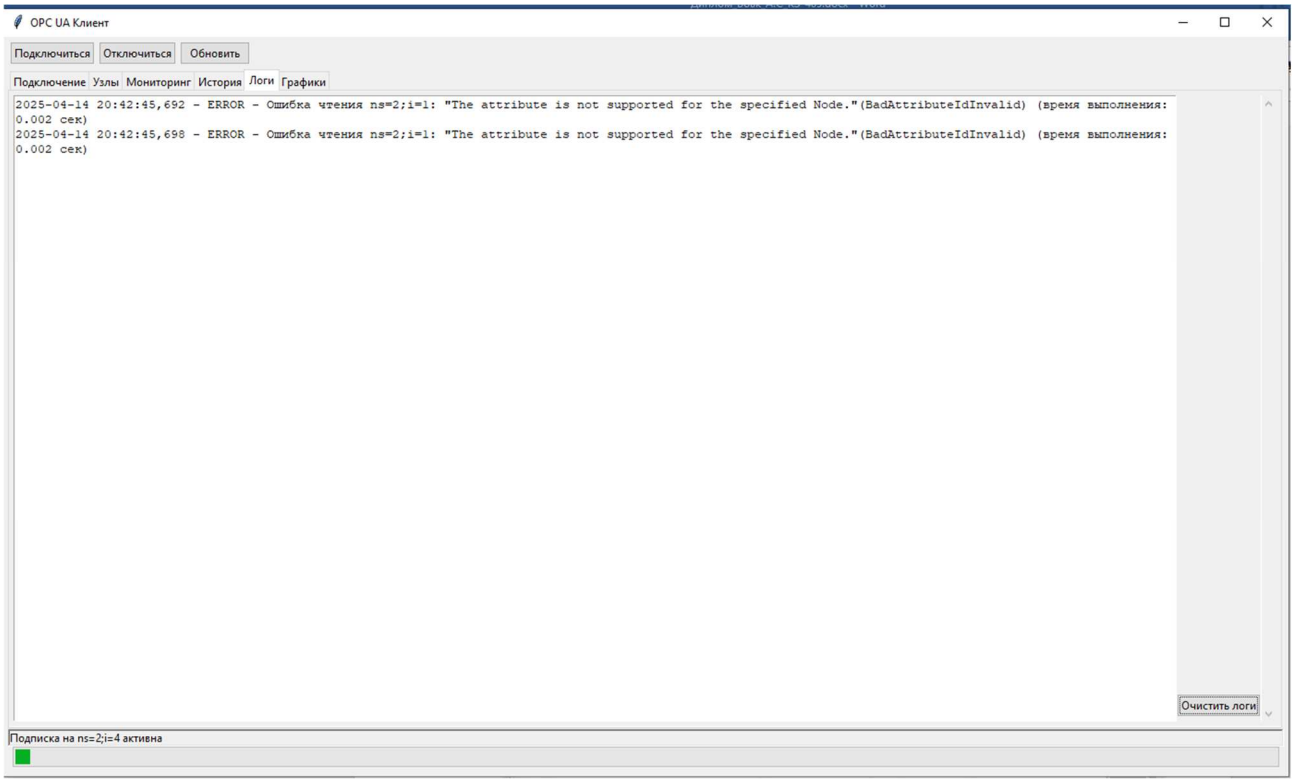

Рисунок 15 – Вкладка «Логи»

ВЫВОД К ГЛАВЕ 3

Был определен стек используемых технологий для реализации клиента, реализована база данных на PostgreSQL для взаимодействия с клиентом. Также был разработан клиент, включающий в себя модуль подключения к базе данных, подключения к OPC UA серверу, чтения и записи содержимого регистров, модуль управления подписками и событиями, а также простой и удобный пользовательский интерфейс.

4. ТЕСТИРОВАНИЕ

Для проверки работоспособности приложения были проведены функциональное и нефункциональное тестирования приложения.

Эти тестирования довольно просты, но позволяют выявить отличия между требуемыми и реально существующими свойствами приложения. Функциональное тестирование позволяет убедиться, что все функции приложения работают корректно и правильно реагируют на входные данные. Нефункциональное тестирование позволяет оценить производительность приложения.

4.1. ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

В таблице 4 приведены результаты функционального тестирования OPC UA клиента и пользовательского интерфейса.

Таблица 4 – результаты функционального тестирования OPC UA клиента и пользовательского интерфейса

Название теста	**Действия**	**Результат**	**Тест пройден?**
Подключение к OPC UA серверу	Подключение к OPC UA серверу по заданному URL, с поддержкой аутентификации по логину и паролю путем ввода и нажатия кнопки «Подключиться»	При нажатии кнопки клиент подключился к OPC UA серверу по заданному URL. При его некорректном вводе клиент выдает ошибку	Да
Отключение от OPC UA сервера	Безопасное отключение от OPC UA сервера путем нажатия кнопки «Отключиться»	При нажатии кнопки клиент безопасно разрывает соединение с OPC UA сервером	Да

Продолжение таблицы 4

Название теста	**Действия**	**Результат**	**Тест пройден?**
Чтение узлов регистров	Ввод ID узла для чтения значения регистра и вывода основной информации о нем, путем нажатия кнопки «Прочитать» в качестве примера был введен ID ns=0; i=2261	При вводе ID узла и нажатии кнопки был прочитан регистр и выведена основная информация о нём. При некорректном написании ID клиент выдает ошибку	Да
Запись узлов регистров	Ввод ID узла для записи регистра и значения, которое необходимо записать и нажатие кнопки «Записать»	При вводе ID узла и значения, и нажатия кнопки клиент записывает значение в регистр. При некорректном вводе клиент выдает ошибку и уведомляет пользователя о возможных ее причинах	Да
Подписка на узлы	Подписка на существующие узлы путем нажатия кнопки «Подписаться»	При нажатии кнопки клиент подписывается на узел. При недостатке прав на подписку клиент выдает ошибку	Да

В таблице 5 приведены результаты функционального тестирования вкладок «Мониторинг» и «История» клиента.

Таблица 5 – Результаты функционального тестирования вкладки «Мониторинг» и «История».

Название теста	**Действия**	**Результат**	**Тест пройден?**
Отписка от узла	Отписка от узла путем нажатия кнопки «Отписаться»	При нажатии кнопки клиент отписывается от существующего узла	Да

Продолжение таблицы 5

Название теста	Действия	Результат	Тест пройден?
Мониторинг событий	При реализации действий, они отображаются в соответствующем окне	При проведении операций подключения к серверу, а также чтения и записи содержимого регистров, события отображаются в соответствующем окне	Да
Фильтрация и сброс параметров истории операций	Фильтрация или сброс истории путем ввода уникальных для каждого события значений и нажатия кнопки «Применить» или «Сбросить»	При вводе соответствующих фильтров и нажатии кнопки, необходимые события отображаются в окне истории	Да

4.2. НЕФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Были установлены минимальные системные требования для приложения.

Чтобы отследить производительность, безопасность, надежность и масштабируемость были использованы библиотеки для тестирования в Python.

Тестирование проводилось на персональном компьютере со следующей конфигурацией:

- процессор Intel(R) Core (TM) i5-12450H , 4400 ГГц;
- 16 ГБ оперативной памяти DDR4;
- видеокарта NVIDIA GeForce RTX 3050, 4 ГБ видеопамяти;
- жесткий диск объемом 512 Гб.

В ходе тестирования, результаты которого представлены на рисунке 16, можно увидеть, что максимальное время чтения узлов 0.002 сек.

```
[CONNECTION] Успешное подключение к opc.tcp://localhost:4840
[READ] Root (i=84): Root...
[READ] Objects (i=85): Objects...
[READ] Server (i=2253): Server...
[READ] CurrentTime (i=2258): 2025-04-26 18:03:36.020158...

[PERFORMANCE] Среднее время: 0.001 сек
[PERFORMANCE] Максимальное время: 0.002 сек
test_1_connection (__main__.OPCUAClientTests.test_1_connection)
Тест подключения к серверу ... ok
test_2_node_reading (__main__.OPCUAClientTests.test_2_node_reading)
Тест чтения узлов ... ok
test_3_performance (__main__.OPCUAClientTests.test_3_performance)
Тест производительности ... ok
test_4_security (__main__.OPCUAClientTests.test_4_security)
Тест параметров безопасности с улучшенной проверкой ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.200s

OK

[SECURITY] Текущая политика безопасности: <opcua.ua.uaprotocol_hand.SecurityPolicy object at 0x000001FE1911CB90>
[WARNING] Нестандартная политика безопасности: <opcua.ua.uaprotocol_hand.SecurityPolicy object at 0x000001FE1911CB90>

[TEARDOWN] Ресурсы освобождены
```

Рисунок 16 – Результаты тестирования клиента

ВЫВОД ПО ГЛАВЕ 4

В четвертом разделе были проведены функциональное и нефункциональное тестирование приложения.

Результаты тестирования показали, что приложение соответствует выдвинутым к нему требованиям.

ЗАКЛЮЧЕНИЕ

В ходе аналитического обзора научно-технической, нормативной и методической литературы по тематике работы было принято решение о разработке OPC UA клиента для считывания регистров и записи регистров с Программируемого логического контроллера. Был проведен сравнительный обзор аналогов, в результате которого было решено, реализовать и клиент на основе OPC UA.NET Client. Были рассмотрены инструменты реализации приложения. Было проведено их сравнение, в итоге которого были выбраны язык программирования Python и Система управления Базами Данных PostgreSQL.

При проектировании архитектуры клиента был проведен анализ проектируемой системы, определены функциональные и нефункциональные требования к клиенту, набор паттернов, используемых в разработке и предполагаемые библиотеки.

Был определен стек используемых технологий для реализации клиента, реализована база данных на PostgreSQL для взаимодействия с клиентом. Также был разработан клиент, включающий в себя модуль подключения к базе данных, подключения к OPC UA серверу, чтения и записи содержимого регистров, модуль управления подписками и событиями, а также простой и удобный пользовательский интерфейс.

Были проведены функциональное и нефункциональное тестирования, согласно результатам которых, приложение соответствует выдвинутым к нему требованиям.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Analysis of OPC UA performances [Электронный ресурс] – URL: https://doi.org/10.1016/j.csi.2013.06.004
2. Антонов, С.В. Проектирование систем автоматизации и управления: Практикум: учебное пособие / С.В. Антонов. – М.: РТУ МИРЭА, 2023 – Ч. 3. – 2023. – 67 с.
3. Базы данных и основы языка SQL: учебное пособие / составитель С.В. Коломийцева. – Хабаровск: ДВГУПС, 2023. – 89 с.
4. Введение в PostgreSQL с Python + psycopg2 [Электронный ресурс] – URL: https://pythonru.com/biblioteki/vvedenie-v-postgresql-s-python-psycopg2?ysclid=m8ebgkcfn3468724455
5. Human Machine Interfaces Based on Open Source Web-Platform and OPC UA [Электронный ресурс] – URL: https://doi.org/10.1016/j.promfg.2020.02.089 (дата обращения: 18.12.2024)
6. Каширская, Е.Н. Основы использования Rexroth ctrlX OPC UA: учебное пособие / Е.Н. Каширская, И.Ю. Зайцев, М.М. Клягин. – М.: РТУ МИРЭА, 2022. – 103 с.
7. Кангин, В.В. Разработка SCADA-систем: учебное пособие / В.В. Кангин, М.В. Кангин, Д.Н. Ямолдинов. – Вологда: Инфра-Инженерия, 2019. – 564 с.
8. Клиент OPC UA сервера UaEpert [Электронный ресурс] – URL: https://www.unified-automation.com/products/development-tools/uaexpert.html
9. Клиент-серверный обмен данными между двумя PLC серии S7-1500 по протоколу OPC UA [Электронный ресурс] – URL: https://habr.com/ru/articles/536038
10. Модуль OPC UA [Электронный ресурс] – URL: https://pythonopcua.readthedocs.io/en/latest/

11. Модуль stdiomask [Электронный ресурс] – URL: https://pypi.org/project/stdiomask/
12. Модуль time в Python [Электронный ресурс] – URL: https://pythonru.com/osnovy/modul-time-v-python
13. Модуль создания графического интерфейса tkinter [Электронный ресурс] – URL: https://docs.python.org/3/library/tkinter.html
14. Описание Multi-Protocol MasterOPC Server. [Электронный ресурс]. – URL: https://insat.ru/products/?category=400
15. Описание стандарта OPC Classic. [Электронный ресурс]. – URL: https://opcfoundation.org/about/opc-technologies/opc-classic/
16. Описание стандарта OPC UA. [Электронный ресурс]. – URL: https://opcfoundation.org/about/opc-technologies/opc-ua/
17. Описание спецификации OPC UA [Электронный ресурс] – URL: https://icscsi.org/library/Documents/ICS_Protocols/OPCF - OPC-UA Part 1 - Overview and Concepts 1.02 Specification.pdf
18. Описание типов классов, используемых OPC UA сервером [Электронный ресурс] – URL: https://documentation.unifiedautomation.com/uasdkhp/1.4.1/html/_l2_ua_node_classes.html
19. Описание устройства адресного пространства OPC UA сервера [Электронный ресурс] – URL: https://documentation.unifiedautomation.com/uasdkhp/1.4.1/html/_l2_ua_address_space_concepts.html
20. OPC UA: новый стандарт в интеграции АСУ ТП / Fine Start. [Электронный ресурс] – URL: https://finestart.school/media/opc_ua
21. Плагин OPC UA Client / MasterOPC [Электронный ресурс] – URL: https://masteropc.ru/uaclient
22. Плагин OPC UA клиент / IntraSCADA, 2024 [Электронный ресурс] – URL: https://docs.intrascada.com/ru/plugin-opcua

23. Просто о стандартах OPC DA и OPC UA / IPC2U, 2023 [Электронный ресурс] – URL: https://ipc2u.ru/articles/prostye-resheniya/prosto-o-standartakh-opc-da-i-opc-ua/
24. Руководство пользователя OPC UA Client & MQTT Publisher Aprotech Kaspersky IoT company [Электронный ресурс] – URL: https://www.aprotech.ru/content/uploads/2024/04/OPC-UA-Client-MQTT-Publisher-User_Guide-RU-1.pdf

ПРИЛОЖЕНИЕ А
МОДУЛЬ ПОДКЛЮЧЕНИЯ К БАЗЕ ДАННЫХ POSTGRESQL

Листинг А.1 – Исходный код DatabaseManager.py

```
import logging

import psycopg2
from psycopg2.extras import DictCursor
from typing import Any, Callable, Optional, Dict, Tuple, List
import datetime


class DatabaseManager:
    """
    Класс для управления подключением и операциями с базой данных PostgreSQL.
    Обеспечивает создание таблиц, проверку структуры БД, логирование событий,
    управление подписками и тегами.
    """

    def __init__(self, db_config: dict):
        """Инициализация менеджера БД с конфигурацией подключения."""
        self.db_config = db_config
        self.conn = None
        self._connect()           # Установка подключения к БД
        self._create_tables()     # Создание необходимых таблиц
        self._check_table_columns()  # Проверка наличия обязательных столбцов
        self._check_database_structure()  # Проверка общей структуры БД

    def get_tag_info(self, node_id: str) -> Optional[Dict]:
        """
        Получение информации о теге по его имени.

        Args:
            node_id: Имя тега для поиска

        Returns:
            Словарь с информацией о теге или None, если тег не найден
        """
        query = """
        SELECT id, name, data_type, description
        FROM tags
        WHERE name = %s
        LIMIT 1
        """
        try:
            with self.conn.cursor(cursor_factory=DictCursor) as cursor:
                cursor.execute(query, (node_id,))
                result = cursor.fetchone()
                return dict(result) if result else None
        except Exception as e:
            logging.error(f"Ошибка получения информации о теге: {e}")
            return None
```

Продолжение листинга А.1

```
def _check_database_structure(self):
        """
        Проверка наличия всех необходимых таблиц и столбцов в базе данных.
        В случае отсутствия обязательных элементов вызывает создание таблиц за-
ново.
        """
        required_columns = {
            'subscriptions': ['id', 'tag_id', 'node_id', 'active', 'created_at'],
            'tags': ['id', 'name', 'data_type', 'description', 'created_at'],
            'events': ['id', 'tag_id', 'event_type', 'message', 'severity',
'timestamp']
        }

        try:
            with self.conn.cursor() as cursor:
                for table, columns in required_columns.items():
                    # Получаем список существующих столбцов для каждой таблицы
                    cursor.execute(f"""
                        SELECT column_name
                        FROM information_schema.columns
                        WHERE table_name = %s
                    """, (table,))
                    existing_columns = {row[0] for row in cursor.fetchall()}

                    # Проверяем наличие всех обязательных столбцов
                    for col in columns:
                        if col not in existing_columns:
                            raise RuntimeError(f"Отсутствует столбец {col} в
таблице {table}")
        except Exception as e:
            logging.error(f"Ошибка проверки структуры БД: {e}")
            self._create_tables()  # Пересоздаем таблицы при ошибке


    def _check_table_columns(self):
        """
        Проверка наличия обязательного столбца node_id в таблице subscriptions.
        Вызывает исключение, если столбец отсутствует.
        """
        try:
            with self.conn.cursor() as cursor:
                cursor.execute("""
                    SELECT column_name
                    FROM information_schema.columns
                    WHERE table_name='subscriptions' AND column_name='node_id'
                """)
                if not cursor.fetchone():
                    raise RuntimeError("Отсутствует столбец node_id в таблице
subscriptions")
        except Exception as e:
            logging.error(f"Ошибка проверки структуры БД: {e}")
    raise
def _connect(self):
        """Установка подключения к базе данных с параметрами из конфигурации."""
        try:
            self.conn = psycopg2.connect(**self.db_config)
            self.conn.autocommit = False  # Отключаем авто-коммит для ручного
управления транзакциями
            logging.info("Подключение к БД успешно")
```

Продолжение листинга А.1

```
        except Exception as e:
            logging.error(f"Ошибка подключения к БД: {e}")
            raise

    def _create_tables(self):
        """
        Создание необходимых таблиц в базе данных, если они не существуют.
        Включает таблицы: tags, events, subscriptions.
        """
        tables = [
            """CREATE TABLE IF NOT EXISTS tags (
                id SERIAL PRIMARY KEY,
                name TEXT NOT NULL UNIQUE,  # Уникальное имя тега
                data_type TEXT NOT NULL,    # Тип данных тега
                description TEXT,           # Описание тега
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP  # Время создания
            )""",
            """CREATE TABLE IF NOT EXISTS events (
                id SERIAL PRIMARY KEY,
                tag_id INTEGER,             # Ссылка на тег
                event_type    TEXT    NOT    NULL,            #    Тип    события
(read/write/value_changed и др.)
                message TEXT NOT NULL,      # Сообщение/значение события
                severity TEXT NOT NULL,     # Уровень важности события
                timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP  # Время события
            )""",
            """CREATE TABLE IF NOT EXISTS subscriptions (
                id SERIAL PRIMARY KEY,
                tag_id INTEGER REFERENCES tags(id),  # Ссылка на тег
                node_id TEXT NOT NULL,      # Идентификатор узла
                active BOOLEAN DEFAULT TRUE,  # Флаг активности подписки
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  # Время создания
                UNIQUE (tag_id, node_id)    # Уникальная пара тег-узел
            )"""
        ]

        try:
            with self.conn.cursor() as cursor:
                for table in tables:
                    cursor.execute(table)
                self.conn.commit()
        except Exception as e:
            self.conn.rollback()  # Откатываем изменения при ошибке
            logging.error(f"Ошибка создания таблиц: {e}")
            raise
def log_event(self, tag_id: Optional[int], event_type: str,
                  message: str, severity: str) -> None:
        """
        Логирование события в базу данных.

        Args:
            tag_id: ID связанного тега (может быть None)
            event_type: Тип события (например, 'value_changed')
            message: Текст сообщения/значения
            severity: Уровень серьезности события
        """
        try:
            # Форматируем текущее время с миллисекундами
            timestamp        =        datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S.%f")[:-3]
```

Продолжение листинга А.1

```
            query = """
            INSERT INTO events (tag_id, event_type, message, severity, timestamp)
            VALUES (%s, %s, %s, %s, %s)
            """
            with self.conn.cursor() as cursor:
                cursor.execute(query, (tag_id, event_type, message, str(sever-
ity), timestamp))
                self.conn.commit()
        except Exception as e:
            self.conn.rollback()
            logging.error(f"Ошибка записи события: {e}")
            raise

    def get_active_subscriptions(self) -> List[Dict]:
        """
        Получение списка активных подписок с последними значениями тегов.

        Returns:
            Список словарей с информацией о подписках и последних значениях
        """
        query = """
        SELECT s.node_id, t.name as tag_name, e.message as last_value, e.timestamp
        FROM subscriptions s
        JOIN tags t ON s.tag_id = t.id
        LEFT JOIN LATERAL (
            SELECT message, timestamp
            FROM events
            WHERE tag_id = t.id AND event_type = 'value_changed'
            ORDER BY timestamp DESC
            LIMIT 1
        ) e ON true
        WHERE s.active = TRUE
        """
        try:
            with self.conn.cursor(cursor_factory=DictCursor) as cursor:
                cursor.execute(query)
                return [dict(row) for row in cursor.fetchall()]
        except Exception as e:
            logging.error(f"Ошибка получения подписок: {e}")
            return []
    def get_active_subscriptions_with_values(self) -> List[Dict]:
        """
        Получение активных подписок с последними значениями (альтернативная реа-
лизация).
        В текущей реализации запрос не завершен.

        """
        query = """
        SELECT s.node_id, t.name as tag_name, e.message as last_value, e.timestamp
        FROM subscriptions s
        JOIN tags t ON s.tag_id = t.id
        LEFT JOIN (
            SELECT tag_id, message, timestamp
            FROM events
            WHERE event_type = 'value_changed'
            ORDER BY timestamp DESC
        ) e ON t.id = e.tag_id
        WHERE s.active = TRUE
        GROUP BY s.node_id, t.name, e.message, e.timestamp
```

Продолжение листинга А.1

```
        """

    def get_node_operations_history(self, limit: int = 100) -> List[Dict]:
        """
        Получение истории операций с узлами (чтение/запись/изменение значений).

        Args:
            limit: Максимальное количество возвращаемых записей

        Returns:
            Список событий с информацией об операциях
        """
        query = """
        SELECT
            e.timestamp,
            e.event_type as operation_type,
            t.name as node_id,
            e.message as value
        FROM events e
        LEFT JOIN tags t ON e.tag_id = t.id
        WHERE e.event_type IN ('read', 'write', 'value_changed')
        ORDER BY e.timestamp DESC
        LIMIT %s
        """
        try:
            with self.conn.cursor(cursor_factory=DictCursor) as cursor:
                cursor.execute(query, (limit,))
                return [dict(row) for row in cursor.fetchall()]
        except Exception as e:
            logging.error(f"Ошибка получения истории операций: {e}")
            return []
def get_or_create_tag(self, name: str, data_type: str, description: str = "") ->
int:
        """
        Создание нового тега или получение существующего.

        Args:
            name: Имя тега
            data_type: Тип данных тега
            description: Описание тега

        Returns:
            ID тега в базе данных
        """
        query = """
        INSERT INTO tags (name, data_type, description)
        VALUES (%s, %s, %s)
        ON CONFLICT (name) DO UPDATE
        SET data_type = EXCLUDED.data_type,
            description = EXCLUDED.description
        RETURNING id
        """
        try:
            with self.conn.cursor() as cursor:
                cursor.execute(query, (name, data_type, description))
                self.conn.commit()
                return cursor.fetchone()[0]
        except Exception as e:
            self.conn.rollback()
            logging.error(f"Error getting/creating tag: {e}")
```

Продолжение листинга А.1

```
        raise

def get_events(self, limit: int = 100) -> List[Dict[str, Any]]:
    """
    Получение списка событий из базы данных.

    Args:
        limit: Максимальное количество возвращаемых событий

    Returns:
        Список событий с подробной информацией
    """
    query = """
    SELECT
        e.id,
        e.timestamp,
        e.event_type,
        COALESCE(t.name, e.message) as node_id,
        e.message,
        e.severity,
        t.name as tag_name
    FROM events e
    LEFT JOIN tags t ON e.tag_id = t.id
    ORDER BY e.timestamp DESC
    LIMIT %s
    """
    try:
        with self.conn.cursor(cursor_factory=DictCursor) as cursor:
            cursor.execute(query, (limit,))
            return [dict(row) for row in cursor.fetchall()]
    except Exception as e:
        logging.error(f"Ошибка получения событий: {e}")
        return []

def get_subscriptions(self) -> List[Dict[str, Any]]:

    """
    Получение списка активных подписок.

    Returns:
        Список подписок с информацией о тегах и узлах
    """
    query = """
    SELECT s.id, t.name as tag_name, s.node_id, s.active
    FROM subscriptions s
    JOIN tags t ON s.tag_id = t.id
    WHERE s.active = TRUE
    """
    try:
        with self.conn.cursor(cursor_factory=DictCursor) as cursor:
            cursor.execute(query)
            return [dict(row) for row in cursor.fetchall()]
    except Exception as e:
        logging.error(f"Ошибка получения подписок: {e}")
        raise

def create_subscription(self, tag_id: int, node_id: str) -> None:
    """
    Создание новой подписки или активация существующей.
```

Окончание листинга А.1

```
        Args:
            tag_id: ID тега для подписки
            node_id: Идентификатор узла
        """
        query = """
        INSERT INTO subscriptions (tag_id, node_id, active)
        VALUES (%s, %s, TRUE)
        ON CONFLICT (tag_id, node_id) DO UPDATE
        SET active = TRUE
        RETURNING id
        """
        try:
            with self.conn.cursor() as cursor:
                cursor.execute(query, (tag_id, node_id))
                self.conn.commit()
        except Exception as e:
            self.conn.rollback()
            logging.error(f"Ошибка создания подписки: {e}")
            raise
def delete_subscription(self, node_id: str) -> None:
        """
        Деактивация подписки по идентификатору узла.

        Args:
            node_id: Идентификатор узла для отписки
        """
        query = """
        UPDATE subscriptions
        SET active = FALSE
        WHERE node_id = %s
        """
        try:
            with self.conn.cursor() as cursor:
                cursor.execute(query, (node_id,))
                self.conn.commit()
        except Exception as e:
            self.conn.rollback()
            logging.error(f"Ошибка удаления подписки: {e}")
            raise

    def close(self):
        """Закрытие соединения с базой данных."""
        if self.conn:
            self.conn.close()
            logging.info("Соединение с БД закрыто")
```

ПРИЛОЖЕНИЕ Б
МОДУЛЬ OPC UA КЛИЕНТА

Листинг Б.1 – Исходный код main.py

```
class OPCUAClient:
    """
    Класс для работы с OPC UA сервером. Обеспечивает:
    - Подключение/отключение от сервера
    - Чтение/запись значений узлов
    - Подписку на изменения значений
    - Мониторинг производительности
    - Тестирование функциональности
    """

    def __init__(self, db_manager: DatabaseManager):
        """Инициализация клиента OPC UA"""
        self.db = db_manager           # Менеджер базы данных для логирования
        self.client = None             # Экземпляр OPC UA клиента
        self._subscriptions = {}      # Активные подписки {node_id: (subscription,
handle)}
        self._callbacks = {}          # Коллбэки для обработки изменений {node_id:
callback}
        self._event_queue = queue.Queue()  # Очередь событий для асинхронной об-
работки
        self._running = False         # Флаг работы клиента
        self._performance_stats = {  # Статистика производительности
            'read': {'count': 0, 'total_time': 0.0},
            'write': {'count': 0, 'total_time': 0.0}
        }

    # ==================== МЕТОДЫ ТЕСТИРОВАНИЯ ====================

    def run_performance_tests(self, node_id: str, iterations=100):
        """
        Тестирование производительности операций чтения.
        Замеряет время одиночных и параллельных запросов.

        Args:
            node_id: Идентификатор узла для тестирования
            iterations: Количество итераций теста

        Returns:
            Словарь с результатами тестирования
        """
        print("\n=== ТЕСТИРОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ===")

        # Тест одиночных запросов
        read_times = []
        for _ in range(iterations):
            start = time.time()
            self.read_node(node_id)
            read_times.append(time.time() - start)

        # Расчет статистики
        avg_read = statistics.mean(read_times)
        max_read = max(read_times)
        print(f"Чтение: Среднее={avg_read:.3f} сек, Макс={max_read:.3f} сек")
        # Тест параллельных запросов (10 потоков)
        with concurrent.futures.ThreadPoolExecutor() as executor:
```

Продолжение листинга Б.1

```
            futures = [executor.submit(self.read_node, node_id) for _ in
range(10)]
            times = []
            for future in concurrent.futures.as_completed(futures):
                times.append(future.result()[1])  # (value, time)

        avg_parallel = statistics.mean(times)
        max_parallel = max(times)
        print(f"Параллельные чтения (10): Среднее={avg_parallel:.3f} сек,
Макс={max_parallel:.3f} сек")

        # Проверка требований производительности
        assert max_read <= 2.0, "Не выполнен критерий производительности для оди-
ночных запросов"
        assert max_parallel <= 2.0, "Не выполнен критерий производительности для
параллельных запросов"

        return {
            'single_read': {'avg': avg_read, 'max': max_read},
            'parallel_read': {'avg': avg_parallel, 'max': max_parallel}
        }

    def run_reliability_tests(self, node_id: str, duration_sec=60):
        """
        Тестирование надежности соединения и операций чтения.

        Args:
            node_id: Идентификатор узла для тестирования
            duration_sec: Длительность теста в секундах

        Returns:
            Словарь с результатами тестирования
        """
        print("\n=== ТЕСТИРОВАНИЕ НАДЕЖНОСТИ ===")

        start_time = time.time()
        successes = 0
        failures = 0

        # Цикл тестирования с заданной длительностью
        while time.time() - start_time < duration_sec:
            try:
                self.read_node(node_id)
                successes += 1
            except Exception:
                failures += 1
            time.sleep(0.1)  # Интервал между запросами

        # Расчет uptime (времени безотказной работы)
        uptime_percent = 100 * successes / (successes + failures)

        print(f"Успешных операций: {successes}, Ошибок: {failures}, Uptime: {up-
time_percent:.2f}%")
        # Проверка требования надежности
        assert uptime_percent >= 99.9, "Не достигнуто 99.9% uptime"
        return {'successes': successes, 'failures': failures, 'uptime': up-
time_percent}

    def test_security(self):
```

Продолжение листинга Б.1

```
        """
        Проверка параметров безопасности соединения.
        Включает проверку режима безопасности и сертификатов.

        Returns:
            Словарь с информацией о безопасности
        """
        print("\n=== ТЕСТИРОВАНИЕ БЕЗОПАСНОСТИ ===")

        if not self.client:
            raise ConnectionError("Нет подключения к серверу")

        # Проверка используемого шифрования
        security_mode = str(self.client.uaclient.security_policy)
        print(f"Режим безопасности: {security_mode}")
        assert "Basic256Sha256" in security_mode, "Не используется 256-битное
шифрование"

        # Проверка сертификата
        cert = self.client.uaclient.secure_channel.security_token.ChannelCertif-
icate
        assert len(cert) > 0, "Сертификат не загружен"
        print(f"Размер сертификата: {len(cert)} байт")

        return {
            'security_mode': security_mode,
            'certificate_size': len(cert)
        }

    def check_write_permission(self, node_id: str) -> bool:
        """
        Проверка прав на запись в указанный узел.

        Args:
            node_id: Идентификатор узла

        Returns:
            True если запись разрешена, False в противном случае
        """
        if not self.client:
            return False

try:
            node = self.client.get_node(node_id)
            try:
                # Пробуем прочитать и записать тестовое значение
                node.get_value()
                node.set_value(ua.Variant(0, ua.VariantType.Int32))
                return True
            except:
                return False

        except ua.uaerrors.BadAttributeIdInvalid:
            logging.warning(f"Узел {node_id} не поддерживает операции чтения/за-
писи")
            return False
        except Exception as e:
            logging.error(f"Ошибка проверки прав для {node_id}: {str(e)}")
```

Продолжение листинга Б.1

```
            return False

    def validate_node_id(self, node_id):
        """
        Валидация формата идентификатора узла.

        Args:
            node_id: Идентификатор узла для проверки

        Returns:
            True если формат корректный, False в противном случае
        """
        try:
            parsed = self._parse_node_id(node_id)
            self.client.get_node(parsed)
            return True
        except:
            return False

    @performance_test
    @reliability_test(retries=3)
    def connect(self, endpoint: str, username: str = None, password: str = None)
-> None:
        """
        Подключение к OPC UA серверу с проверкой параметров.

        Args:
            endpoint: Адрес сервера (формат: opc.tcp://host:port)
            username: Имя пользователя (опционально)
            password: Пароль (опционально)

        Raises:
            ConnectionError: При ошибках подключения
        """
        try:
            # Валидация формата endpoint
            if not endpoint.startswith("opc.tcp://"):
                raise ValueError("Endpoint должен начинаться с 'opc.tcp://'")

            # Разбор host и port
            host_port = endpoint.replace("opc.tcp://", "").split(":")
            if len(host_port) != 2:
                raise   ValueError("Неверный   формат   endpoint.   Ожидается:
opc.tcp://host:port")
            # Проверка доступности сервера через сокет
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                sock.settimeout(5)
                sock.connect((host_port[0], int(host_port[1])))

            # Инициализация клиента
            if hasattr(self, 'client') and self.client:
                self.disconnect()

            self.client = Client(endpoint, timeout=5)

            # Настройка аутентификации
            if username and password:
                self.client.set_user(username)
                self.client.set_password(password)
```

Продолжение листинга Б.1

```
                # Хеширование пароля для логов
                pass_hash = hashlib.sha256(password.encode()).hexdigest()[:8]
                self.db.log_event(
                    None,
                    "auth",
                    f"Аутентификация как {username} (passhash: {pass_hash})",
                    EventSeverity.MEDIUM.value
                )

            self.client.connect()
            self._running = True

            # Запуск обработчика событий в отдельном потоке
            threading.Thread(
                target=self._process_events,
                daemon=True,
                name="OPCUA_Event_Processor"
            ).start()

            logging.info(f"Успешное подключение к {endpoint}")
            self.db.log_event(
                None,
                "connection",
                f"Подключено к {endpoint} как {username if username else
'аноним'}",
                EventSeverity.MEDIUM.value
            )

            # Запуск мониторинга производительности
            self._start_performance_monitor()

            return True

        except socket.timeout as e:
            error_msg = f"Таймаут подключения к {endpoint}"
            logging.error(error_msg)
            self.db.log_event(None, "connection_timeout", error_msg, EventSever-
ity.HIGH.value)
            raise ConnectionError(error_msg) from e
        except Exception as e:
            error_msg = f"Ошибка подключения к {endpoint}: {str(e)}"
            logging.error(error_msg, exc_info=True)
            self.db.log_event(None, "connection_error", error_msg, EventSever-
ity.HIGH.value)
            raise ConnectionError(error_msg) from e

    def _start_performance_monitor(self):
        """
        Запуск фонового мониторинга производительности.
        Анализирует статистику операций и генерирует предупреждения.
        """
        def monitor():
            while self._running:
                try:
                    # Интервал проверки - 10 секунд
                    time.sleep(10)

                    # Анализ статистики чтения
                    if self._performance_stats['read']['count'] > 0:
```

Продолжение листинга Б.1

```
                        avg_read = self._performance_stats['read']['total_time']
/ self._performance_stats['read'][
                            'count']
                        if avg_read > 1.0:  # Порог для предупреждения
                            logging.warning(f"Высокое среднее время чтения:
{avg_read:.3f} сек")
                            self.db.log_event(
                                None,
                                "performance_warning",
                                f"Высокое время чтения: {avg_read:.3f} сек",
                                EventSeverity.MEDIUM.value
                            )

                    # Сброс статистики
                    self._performance_stats = {
                        'read': {'count': 0, 'total_time': 0.0},
                        'write': {'count': 0, 'total_time': 0.0}
                    }
                except Exception as e:
                    logging.error(f"Ошибка монитора производительности:
{str(e)}")

        threading.Thread(target=monitor, daemon=True).start()

    def _parse_node_id(self, node_id):

        """
        Парсинг идентификатора узла в стандартный формат.

        Args:
            node_id: Идентификатор узла (число или строка)

        Returns:
            Стандартизированный идентификатор узла
        """
        if isinstance(node_id, int):
            return f"ns=2;i={node_id}"
        elif isinstance(node_id, str):
            if not any(p in node_id for p in ['ns=', 'i=', 's=']):
                return f"ns=2;i={node_id}"  # Автоматическое добавление namespace
        return node_id

    @performance_test
    @reliability_test(retries=2)
    def read_node(self, node_id: str, timeout: float = 5.0) -> Any:
        """
        Чтение значения узла с обработкой различных типов узлов.

        Args:
            node_id: Идентификатор узла
            timeout: Таймаут операции

        Returns:
            Значение узла или его атрибутов

        Raises:
            ConnectionError: При проблемах с подключением
            ValueError: При ошибках чтения
        """
```

Продолжение листинга Б.1

```
        try:
            node = self.client.get_node(node_id)

            # Попытка чтения значения
            try:
                return node.get_value()
            except ua.uaerrors.BadAttributeIdInvalid:
                # Если узел не поддерживает чтение Value, пробуем другие атрибуты
                try:
                    return node.get_display_name().Text
                except:
                    return node.get_browse_name().Name

        except Exception as e:
            logging.error(f"Ошибка чтения узла {node_id}: {str(e)}")
            raise
    def subscribe(self, node_id: str, callback: Optional[Callable] = None) ->
None:
        """
        Подписка на изменения значения узла.
        Args:
            node_id: Идентификатор узла
            callback: Функция обратного вызова при изменении

        Raises:
            ConnectionError: Если нет подключения
            ValueError: При ошибках подписки
        """
        if not self.client:
            raise ConnectionError("Нет подключения к серверу")
        try:
            node = self.client.get_node(node_id)
            try:
                node.get_value()
            except ua.uaerrors.BadAttributeIdInvalid:
                raise ValueError(f"Узел {node_id} не поддерживает подписки")

            # Создание новой подписки если ее нет
            if node_id not in self._subscriptions:
                handler = SubHandler(self._handle_subscription)
                subscription = self.client.create_subscription(1000, handler)
                handle = subscription.subscribe_data_change(node)

                self._subscriptions[node_id] = (subscription, handle)
                self._callbacks[node_id] = callback or (lambda n, v: None)

                # Регистрация подписки в БД
                try:
                    tag_id = self.db.get_or_create_tag(node_id, "unknown")
                    self.db.create_subscription(tag_id, node_id)
                except Exception as db_error:
                    logging.error(f"Ошибка БД при подписке: {db_error}")
                    raise

        except Exception as e:
            error_msg = f"Ошибка подписки на {node_id}: {str(e)}"
            logging.error(error_msg)
            raise ValueError(error_msg)
```

Продолжение листинга Б.1

```
def _check_node_supports_subscription(self, node) -> bool:
    """
    Проверка поддержки подписки узлом.

    Args:
        node: Объект узла OPC UA

    Returns:
        True если узел поддерживает подписку
    """
    try:
        node.get_value()
        return True
    except ua.uaerrors.BadAttributeIdInvalid:
        return False
    except Exception as e:
        logging.warning(f"Ошибка проверки узла: {str(e)}")
        return False

def _validate_node_id(self, node_id: str) -> bool:
    """
    Валидация формата идентификатора узла по регулярным выражениям.

    Args:
        node_id: Идентификатор узла
    Returns:
        True если формат корректный
    """
    patterns = [
        r'^ns=\d+;i=\d+$',  # Числовой идентификатор
        r'^ns=\d+;s=\w+$',  # Строковый идентификатор
        r'^ns=\d+;g=[0-9a-f-]+$'  # GUID идентификатор
    ]
    return any(re.fullmatch(p, node_id) for p in patterns)

def _handle_subscription(self, node_id: str, value: Any) -> None:
    """
    Обработчик изменений подписанных узлов.
    Помещает события в очередь для асинхронной обработки.
    """
    self._event_queue.put(("datachange", node_id, value))

def _process_events(self) -> None:
    """
    Обработчик событий из очереди.
    Работает в отдельном потоке пока _running = True.
    """
    while self._running:
        try:
            event_type, *args = self._event_queue.get(timeout=1)
            if event_type == "datachange":
                node_id, value = args
                # Вызов коллбэка если он зарегистрирован
                if node_id in self._callbacks:
                    self._callbacks[node_id](node_id, value)

                # Логирование события в БД
                tag_id = self.db.get_or_create_tag(node_id, "unknown")
```

Продолжение листинга Б.1

```
                self.db.log_event(
                    tag_id,
                    "value_changed",
                    f"Значение изменилось на {value}",
                    EventSeverity.LOW.value
                )

        except queue.Empty:
            continue

def write_node(self, node_id: str, value: Any, timeout: float = 5.0) -> None:
    """
    Запись значения в узел с проверкой прав и обработкой таймаута.

    Args:
        node_id: Идентификатор узла
        value: Значение для записи
        timeout: Таймаут операции

    Raises:
        ConnectionError: При проблемах с подключением
        PermissionError: При отсутствии прав
        TimeoutError: При превышении таймаута
    """
    if not self.client:
        raise ConnectionError("Нет подключения к серверу")

    try:
        node = self.client.get_node(node_id)
        if not node:
            raise ValueError(f"Узел {node_id} не существует")
    except ua.uaerrors.BadNodeIdUnknown:
        raise ValueError(f"Узел {node_id} не найден на сервере")

    # Проверка прав на запись
    if not self.check_write_permission(node_id):
        raise PermissionError(f"Нет прав на запись в узел {node_id}")

    result = None
    error = None
    event = threading.Event()

    def write_worker():
        """Функция для выполнения записи в отдельном потоке"""
        nonlocal result, error
        try:
            # Определение типа значения и создание Variant
            if isinstance(value, bool):
                variant = ua.Variant(value, ua.VariantType.Boolean)
            elif isinstance(value, int):
                variant = ua.Variant(value, ua.VariantType.Int32)
            elif isinstance(value, float):
                variant = ua.Variant(value, ua.VariantType.Float)
            elif isinstance(value, str):
                variant = ua.Variant(value, ua.VariantType.String)
            else:
                variant = ua.Variant(value)

            node.set_value(variant)
```

Продолжение листинга Б.1

```
                result = True
                # Логирование успешной записи
                tag_id = self.db.get_or_create_tag(node_id, type(value).__name__)
                self.db.log_event(
                    tag_id,
                    "write_success",
                    f"Значение {value} записано",
                    EventSeverity.MEDIUM.value
                )

            except ua.uaerrors.BadUserAccessDenied as e:
                error = PermissionError(f"Доступ запрещен: {str(e)}")
            except Exception as e:
                error = e
            finally:
                event.set()
        # Запуск записи в отдельном потоке
        thread = threading.Thread(target=write_worker, daemon=True)
        thread.start()

        # Ожидание завершения с таймаутом
        if not event.wait(timeout=timeout):
            error = TimeoutError(f"Таймаут записи в узел {node_id}")
            try:
                self.client.disconnect()
            except:
                pass

        # Обработка ошибок
        if error:
            error_msg = f"Ошибка записи в {node_id}: {str(error)}"
            logging.error(error_msg)
            self.db.log_event(None,   "write_error",   error_msg,   EventSever-
ity.HIGH.value)
            raise error

    def check_node_attributes(self, node_id: str):
        """
        Вывод атрибутов узла для отладки.

        Args:
            node_id: Идентификатор узла
        """
        try:
            node = self.client.get_node(node_id)
            print(f"\nАтрибуты узла {node_id}:")
            for attr in ua.AttributeIds:
                try:
                    value = node.get_attribute(attr)
                    print(f"{attr.name}: {value.Value.Value}")
                except:
                    pass
        except Exception as e:
            print(f"Ошибка проверки атрибутов: {str(e)}")

    def node_exists(self, node_id: str, timeout: float = 2.0) -> bool:
        """
        Проверка существования узла.
```

Продолжение листинга Б.1

```
        Args:
            node_id: Идентификатор узла
            timeout: Таймаут операции

        Returns:
            True если узел существует
        """
        if not self.client:
            return False

        result = False
        event = threading.Event()

        def check_worker():
            """Функция проверки в отдельном потоке"""
            nonlocal result
            try:
                node = self.client.get_node(node_id)
                node.get_browse_name()  # Простая проверка доступности
                result = True
            except:
                result = False
            finally:
                event.set()

        thread = threading.Thread(target=check_worker, daemon=True)
        thread.start()

        event.wait(timeout)
        return result

    def safe_read_node(self, node_id: str, retries: int = 2, timeout: float = 5.0)
-> Any:
        """
        Безопасное чтение узла с повторными попытками при ошибках.

        Args:
            node_id: Идентификатор узла
            retries: Количество попыток
            timeout: Таймаут каждой попытки

        Returns:
            Значение узла

        Raises:
            ConnectionError: При проблемах с подключением
            ValueError: При ошибках чтения
        """
        for attempt in range(retries):
            try:
                return self.read_node(node_id, timeout)
            except (ConnectionError, TimeoutError) as e:
                if attempt == retries - 1:
                    raise
                logging.warning(f"Попытка {attempt + 1}: переподключение...")
                self.reconnect()
            except ua.uaerrors.BadAttributeIdInvalid:
                try:
                    node = self.client.get_node(node_id)
```

Продолжение листинга Б.1

```
                return node.get_display_name().Text
            except:
                raise ValueError(f"Не удалось прочитать узел {node_id}")

    def reconnect(self):
        """Переподключение к серверу с текущими параметрами."""
        try:
            if self.client:
                self.client.disconnect()
            self.client = Client(self.endpoint)
            if hasattr(self, 'username') and hasattr(self, 'password'):
                self.client.set_user(self.username)
                self.client.set_password(self.password)
            self.client.connect()
            logging.info("Переподключение к серверу успешно")
        except Exception as e:
            logging.error(f"Ошибка переподключения: {str(e)}")
            raise ConnectionError(f"Не удалось переподключиться: {str(e)}")

    def find_available_nodes(self, start_node="i=84"):
        """
        Поиск доступных узлов начиная с указанного.

        Args:
            start_node: Начальный узел для поиска (по умолчанию Root)

        Returns:
            Список кортежей (node_id, browse_name)
        """
        available = []
        try:
            node = self.client.get_node(start_node)
            children = node.get_children()

            for child in children[:10]:  # Ограничение количества для теста
                try:
                    browse_name = child.get_browse_name().Name
                    node_id = child.nodeid.to_string()
                    available.append((node_id, browse_name))
                except:
                    continue
        except Exception as e:
            logging.error(f"Ошибка поиска узлов: {str(e)}")

        return available

    def browse_node(self, node_id: str = "i=84"):
        """
        Вывод дочерних узлов для указанного узла.

        Args:
            node_id: Идентификатор родительского узла
        """
        try:
            node = self.client.get_node(node_id)
            print(f"\nДочерние узлы {node_id}:")
            for child in node.get_children():
                print(f"{child.nodeid.to_string()}                                    -
{child.get_browse_name().Name}")
```

Продолжение листинга Б.1

```
            except Exception as e:
                print(f"Ошибка просмотра узлов: {str(e)}")

    def _determine_variant_type(self, value: Any) -> ua.VariantType:
            """
            Определение типа Variant для значения.

            Args:
                value: Значение для определения типа

            Returns:
                Соответствующий VariantType
            """
            if isinstance(value, bool):
                return ua.VariantType.Boolean
            elif isinstance(value, int):
                return ua.VariantType.Int64 if value > 2147483647 else ua.Vari-
antType.Int32
            elif isinstance(value, float):
                return ua.VariantType.Double
            elif isinstance(value, str):
                return ua.VariantType.String
            elif isinstance(value, datetime):
                return ua.VariantType.DateTime
            return ua.VariantType.Null

    def disconnect(self) -> None:
            """Отключение от сервера с очисткой подписок."""
            if not self.client:
                return

            try:
                self._running = False
                # Удаление всех подписок
                for node_id, (subscription, _) in list(self._subscriptions.items()):
                    try:
                        subscription.delete()
                        self.db.delete_subscription(node_id)
                    except Exception as e:
                        logging.warning(f"Ошибка отписки: {str(e)}")

                self._subscriptions.clear()
                self._callbacks.clear()
                self.client.disconnect()
                self.client = None

                logging.info("Отключено от сервера")
                self.db.log_event(
                    None,
                    "disconnection",
                    "Отключено от сервера",
                    EventSeverity.MEDIUM.value
                )
            except Exception as e:
                error_msg = f"Ошибка отключения: {str(e)}"
                logging.error(error_msg)
                self.db.log_event(
                    None,
                    "disconnection_error",
```

Окончание листинга Б.1

```
            error_msg,
            EventSeverity.HIGH.value
        )
        raise

def unsubscribe(self, node_id: str) -> None:
    """
    Отписка от изменений узла.

    Args:
        node_id: Идентификатор узла

    Raises:
        ValueError: Если подписка не найдена или ошибка отписки
    """
    if node_id not in self._subscriptions:
        raise ValueError(f"Нет активной подписки на узел {node_id}")
    try:
        subscription, handle = self._subscriptions[node_id]
        subscription.unsubscribe(handle)
        del self._subscriptions[node_id]
        del self._callbacks[node_id]

        self.db.delete_subscription(node_id)

        logging.info(f"Отписались от узла {node_id}")
    except Exception as e:
        error_msg = f"Ошибка отписки от узла {node_id}: {str(e)}"
        logging.error(error_msg)
        raise ValueError(error_msg)
```

ПРИЛОЖЕНИЕ В
РЕАЛИЗАЦИЯ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

Листинг В.1 – исходный код OPCUAClientUI.py

```
import tkinter as tk
from tkinter import ttk, messagebox
import logging
import threading
import DatabaseManager
import stdiomask
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import time


# ==================== ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ====================
class OPCUAClientGUI:
    """Главный класс графического интерфейса OPC UA клиента"""

    def __init__(self, root: tk.Tk):
        """Инициализация главного окна приложения"""
        self.root = root
        self.root.title("OPC UA Клиент")
        self.root.geometry("1200x800")
        # Обработчики закрытия окна
        self.root.protocol("WM_DELETE_WINDOW", self._on_close)
        self.root.protocol("WM_DELETE_WINDOW", self._safe_exit)
        # Запуск автообновления событий
        self._auto_refresh_events()

        try:
            # Подключение к базе данных
            self.db = DatabaseManager.DatabaseManager({
                'host': 'localhost',
                'port': '5432',
                'dbname': 'OPC_UA',
                'user': 'postgres',
                'password': 'postgres'
            })
        except Exception as e:
            messagebox.showerror("Ошибка БД", f"Не удалось подключиться:
{str(e)}")
            self.root.destroy()
            return

        from main import OPCUAClient
        # Создание OPC UA клиента
        self.opcua_client = OPCUAClient(self.db)
        # Настройка интерфейса
        self._setup_ui()

    def _apply_history_filter(self):
        """Применение фильтра к истории операций"""
        if not hasattr(self, 'history_tree'):
            return
        filter_text = self.history_filter.get().lower()
        # Скрытие несоответствующих фильтру записей
        for item in self.history_tree.get_children():
            values = self.history_tree.item(item)['values']
```

Продолжение листинга В.1

```
            if values and not any(filter_text in str(v).lower() for v in values
if v):
                self.history_tree.detach(item)

    def _reset_history_filter(self):
        """Сброс фильтра истории операций"""
        self.history_filter.delete(0, tk.END)
        # Показ всех скрытых записей
        for item in self.history_tree.get_children():
            self.history_tree.move(item, "", "end")

    def _auto_refresh_events(self):
        """Автоматическое обновление событий и графика"""
        self._refresh_events()

        if hasattr(self, 'graph_node_entry') and self.graph_node_entry.get():
            try:
                self._plot_graph()
            except Exception as e:
                logging.error(f"Ошибка автообновления графика: {str(e)}")

        # Планирование следующего обновления через 30 секунд
        self.root.after(30000, self._auto_refresh_events)

    def _safe_exit(self):
        """Безопасное завершение работы приложения"""
        try:
            if self.opcua_client and self.opcua_client.client:
                self.opcua_client.disconnect()
            self.db.close()
        except Exception as e:
            logging.error(f"Ошибка при завершении: {e}")
        finally:
            self.root.destroy()

    def _on_close(self):
        """Обработчик закрытия окна"""
        try:
            if self.opcua_client:
                self.opcua_client.disconnect()
            self.db.close()
        except Exception as e:
            logging.error(f"Ошибка при закрытии: {e}")
        finally:
            self.root.destroy()
    def _setup_ui(self):
        """Настройка основного интерфейса"""
        self._create_main_container()
        self._create_toolbar()
        self._create_notebook()
        self._create_status_bar()
        # Создание вкладки с графиками
        self.graph_tab = ttk.Frame(self.notebook)
        self._setup_graph_tab()
        self.notebook.add(self.graph_tab, text="Графики")

    def _setup_graph_tab(self):
        """Настройка вкладки с графиками"""
        frame = ttk.Frame(self.graph_tab)
```

Продолжение листинга В.1

```
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Элементы управления графиком
        ttk.Label(frame, text="Node ID для графика:").grid(row=0, column=0,
sticky=tk.W)
        self.graph_node_entry = ttk.Entry(frame, width=40)
        self.graph_node_entry.grid(row=0, column=1, sticky=tk.W, padx=5)

        ttk.Label(frame, text="Период (часы):").grid(row=1, column=0,
sticky=tk.W)
        self.graph_hours_entry = ttk.Entry(frame, width=10)
        self.graph_hours_entry.insert(0, "24")  # Значение по умолчанию
        self.graph_hours_entry.grid(row=1, column=1, sticky=tk.W, padx=5)

        self.plot_btn = ttk.Button(
            frame,
            text="Построить график",
            command=self._plot_graph
        )
        self.plot_btn.grid(row=1, column=2, padx=5)

        # Область для отображения графика
        self.graph_frame = ttk.Frame(frame)
        self.graph_frame.grid(row=2, column=0, columnspan=3, sticky="nsew",
pady=10)

        frame.columnconfigure(0, weight=1)
        frame.rowconfigure(2, weight=1)

    def _plot_graph(self):
        """Построение графика значений тега"""
        node_id = self.graph_node_entry.get().strip()
        hours = self.graph_hours_entry.get().strip()

        if not node_id:
            messagebox.showerror("Ошибка", "Введите Node ID для построения
графика")
            return

        try:
            hours = float(hours)
            if hours <= 0:
                raise ValueError("Период должен быть положительным числом")
        except ValueError:
            messagebox.showerror("Ошибка", "Некорректное значение периода")

            return

        try:
            # Запрос данных из базы
            query = """
            SELECT timestamp, message
            FROM events
            WHERE tag_id = (SELECT id FROM tags WHERE name = %s)
              AND event_type = 'value_changed'
              AND timestamp >= NOW() - INTERVAL '%s hours'
            ORDER BY timestamp
            """
```

Продолжение листинга В.1

```
            with self.db.conn.cursor() as cursor:
                cursor.execute(query, (node_id, hours))
                data = cursor.fetchall()

            if not data:
                messagebox.showinfo("Информация", "Нет данных для построения гра-
фика")
                return

            # Подготовка данных для графика
            timestamps = [row[0] for row in data]
            values = []
            for row in data:
                try:
                    values.append(float(row[1]))
                except ValueError:
                    values.append(0)

            # Создание графика
            fig, ax = plt.subplots(figsize=(10, 4))
            ax.plot(timestamps, values, '-o', label=node_id)
            ax.set_xlabel('Время')
            ax.set_ylabel('Значение')
            ax.set_title(f'График значений тега {node_id}')
            ax.grid(True)
            ax.legend()

            # Очистка предыдущего графика
            for widget in self.graph_frame.winfo_children():
                widget.destroy()

            # Отображение графика в интерфейсе
            canvas = FigureCanvasTkAgg(fig, master=self.graph_frame)
            canvas.draw()
            canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        except Exception as e:
            logging.error(f"Ошибка построения графика: {str(e)}")
            messagebox.showerror("Ошибка",   f"Не   удалось   построить   график:
{str(e)}")

    def _setup_connection_tab(self):
        """Настройка вкладки подключения"""
        frame = ttk.LabelFrame(self.connection_tab, text="Параметры подключения",
padding=10)
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Поля ввода параметров подключения
        ttk.Label(frame, text="Endpoint:").grid(row=0, column=0, sticky=tk.W)
        self.endpoint_entry = ttk.Entry(frame, width=40)
        self.endpoint_entry.insert(0, "opc.tcp://localhost:4840")
        self.endpoint_entry.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=2)

        ttk.Label(frame, text="Логин:").grid(row=1, column=0, sticky=tk.W)
        self.username_entry = ttk.Entry(frame, width=20)
        self.username_entry.grid(row=1, column=1, sticky=tk.W, padx=5, pady=2)

        ttk.Label(frame, text="Пароль:").grid(row=2, column=0, sticky=tk.W)
        self.password_entry = ttk.Entry(frame, width=20, show="*")
```

Продолжение листинга В.1

```
        self.password_entry.grid(row=2, column=1, sticky=tk.W, padx=5, pady=2)

        self.secure_pwd_btn = ttk.Button(
            frame,
            text="Безопасный ввод пароля",
            command=self._secure_password_input
        )
        self.secure_pwd_btn.grid(row=2, column=2, padx=5)

        frame.columnconfigure(1, weight=1)

    def _secure_password_input(self):
        """Безопасный ввод пароля через консоль"""
        try:
            password = stdiomask.getpass(prompt="Введите пароль: ", mask='*')
            self.password_entry.delete(0, tk.END)
            self.password_entry.insert(0, password)
        except Exception as e:
            logging.error(f"Ошибка ввода пароля: {str(e)}")
            messagebox.showerror("Ошибка", "Не удалось выполнить безопасный ввод
пароля")

    def _create_main_container(self):
        """Создание основного контейнера"""

        self.main_frame = ttk.Frame(self.root)
        self.main_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

    def _create_toolbar(self):
        """Создание панели инструментов"""
        self.toolbar = ttk.Frame(self.main_frame)
        self.toolbar.pack(fill=tk.X, pady=(0, 5))

        # Кнопки toolbar
        self.connect_btn = ttk.Button(self.toolbar, text="Подключиться", com-
mand=self._connect)
        self.connect_btn.pack(side=tk.LEFT, padx=2)

        self.disconnect_btn = ttk.Button(self.toolbar, text="Отключиться", com-
mand=self._disconnect, state=tk.DISABLED)
        self.disconnect_btn.pack(side=tk.LEFT, padx=2)

        self.refresh_btn = ttk.Button(self.toolbar, text="Обновить", com-
mand=self._refresh_data)
        self.refresh_btn.pack(side=tk.LEFT, padx=2)

    def _create_notebook(self):
        """Создание набора вкладок"""
        self.notebook = ttk.Notebook(self.main_frame)
        self.notebook.pack(fill=tk.BOTH, expand=True)

        # Создание вкладок
        self.connection_tab = ttk.Frame(self.notebook)
        self._setup_connection_tab()
        self.notebook.add(self.connection_tab, text="Подключение")

        self.nodes_tab = ttk.Frame(self.notebook)
        self._setup_nodes_tab()
        self.notebook.add(self.nodes_tab, text="Узлы")
```

Продолжение листинга В.1

```
        self.monitor_tab = ttk.Frame(self.notebook)
        self._setup_monitor_tab()
        self.notebook.add(self.monitor_tab, text="Мониторинг")

        self.history_tab = ttk.Frame(self.notebook)
        self._setup_history_tab()
        self.notebook.add(self.history_tab, text="История")

        self.logs_tab = ttk.Frame(self.notebook)
        self._setup_logs_tab()
        self.notebook.add(self.logs_tab, text="Логи")

    def _create_status_bar(self):
        """Создание строки состояния"""
        self.progress_bar = ttk.Progressbar(
            self.main_frame,
            mode='indeterminate',
            length=200
        )
        self.progress_bar.pack(side=tk.BOTTOM, fill=tk.X, padx=5, pady=(0, 5))

        self.status_var = tk.StringVar(value="Готов")
        self.status_bar = ttk.Label(
            self.main_frame,
            textvariable=self.status_var,
            relief=tk.SUNKEN,
            anchor=tk.W
        )
        self.status_bar.pack(side=tk.BOTTOM, fill=tk.X)

    def _setup_nodes_tab(self):
        """Настройка вкладки работы с узлами"""
        frame = ttk.LabelFrame(self.nodes_tab, text="Управление узлами", pad-
ding=10)
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Поля для работы с узлом
        ttk.Label(frame, text="Node ID:").grid(row=0, column=0, sticky=tk.W)
        self.node_id_entry = ttk.Entry(frame, width=40)
        self.node_id_entry.grid(row=0, column=1, columnspan=2, sticky=tk.EW,
padx=5, pady=2)

        self.read_btn = ttk.Button(frame, text="Прочитать", com-
mand=self._read_node, state=tk.DISABLED)
        self.read_btn.grid(row=0, column=3, padx=5)

        ttk.Label(frame, text="Значение:").grid(row=1, column=0, sticky=tk.W)
        self.value_entry = ttk.Entry(frame, width=40)
        self.value_entry.grid(row=1, column=1, columnspan=2, sticky=tk.EW,
padx=5, pady=2)

        self.write_btn = ttk.Button(frame, text="Записать", com-
mand=self._write_node, state=tk.DISABLED)
        self.write_btn.grid(row=1, column=3, padx=5)

        # Область с информацией о теге
        tag_info_frame = ttk.LabelFrame(frame, text="Информация о теге", pad-
ding=5)
        tag_info_frame.grid(row=2, column=0, columnspan=4, sticky=tk.EW, pady=5)
```

Продолжение листинга В.1

```
        self.tag_info_text = tk.Text(
            tag_info_frame,
            height=4,
            state=tk.DISABLED,
            wrap=tk.WORD,
            font=('Arial', 10)
        )
        scrollbar          =          ttk.Scrollbar(tag_info_frame,          com-
mand=self.tag_info_text.yview)
        self.tag_info_text.configure(yscrollcommand=scrollbar.set)

        self.tag_info_text.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        self.subscribe_btn = ttk.Button(
            frame,
            text="Подписаться",
            command=self._subscribe_node,
            state=tk.DISABLED
        )
        self.subscribe_btn.grid(row=3, column=3, padx=5, pady=5)

        frame.columnconfigure(1, weight=1)
    def _setup_monitor_tab(self):
        """Настройка вкладки мониторинга"""
        frame = ttk.Frame(self.monitor_tab)
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Область активных подписок
        subs_frame = ttk.LabelFrame(frame, text="Активные подписки", padding=5)
        subs_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        self.subscriptions_tree = ttk.Treeview(
            subs_frame,
            columns=("node_id", "tag_name", "value", "timestamp"),
            show="headings",
            height=8
        )
        self.subscriptions_tree.heading("node_id", text="Node ID")
        self.subscriptions_tree.heading("tag_name", text="Имя тега")
        self.subscriptions_tree.heading("value", text="Значение")
        self.subscriptions_tree.heading("timestamp",             text="Последнее
обновление")

        self.subscriptions_tree.bind('<<TreeviewSelect>>',    self._on_subscrip-
tion_select)

        scrollbar    =    ttk.Scrollbar(subs_frame,    orient="vertical",    com-
mand=self.subscriptions_tree.yview)
        self.subscriptions_tree.configure(yscrollcommand=scrollbar.set)

        self.subscriptions_tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        btn_frame = ttk.Frame(subs_frame)
        btn_frame.pack(fill=tk.X, pady=5)

        self.unsubscribe_btn = ttk.Button(
            btn_frame,
```

Продолжение листинга В.1

```
            text="Отписаться",
            command=self._unsubscribe,
            state=tk.DISABLED
        )

        self.unsubscribe_btn.pack(side=tk.LEFT, padx=5)

        ttk.Button(
            btn_frame,
            text="Обновить",
            command=self._refresh_subscriptions
        ).pack(side=tk.RIGHT, padx=5)

        # Область событий
        events_frame = ttk.LabelFrame(frame, text="События", padding=5)
        events_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        self.events_tree = ttk.Treeview(
            events_frame,
            columns=("timestamp", "event_type", "node_id", "message", "sever-
ity"),
            show="headings",
            height=8
        )
        self.events_tree.heading("timestamp", text="Время")
        self.events_tree.heading("event_type", text="Тип события")
        self.events_tree.heading("node_id", text="Node ID")
        self.events_tree.heading("message", text="Сообщение")
        self.events_tree.heading("severity", text="Уровень")

        scrollbar = ttk.Scrollbar(events_frame, orient="vertical", com-
mand=self.events_tree.yview)
        self.events_tree.configure(yscrollcommand=scrollbar.set)

        self.events_tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        btn_frame = ttk.Frame(events_frame)
        btn_frame.pack(fill=tk.X, pady=5)

        ttk.Button(
            btn_frame,
            text="Обновить",
            command=self._refresh_events
        ).pack(side=tk.RIGHT, padx=5)

        # Первоначальное обновление данных
        self._refresh_subscriptions()
        self._refresh_events()

    def _on_subscription_select(self, event):
        """Обработчик выбора подписки в дереве"""
        selected = self.subscriptions_tree.selection()
        self.unsubscribe_btn['state'] = tk.NORMAL if selected else tk.DISABLED

    def _setup_history_tab(self):
        """Настройка вкладки истории операций"""

        frame = ttk.Frame(self.history_tab)
```

Продолжение листинга В.1

```
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Таблица истории операций
        self.history_tree = ttk.Treeview(
            frame,
            columns=("timestamp", "operation", "node_id", "value"),
            show="headings",
            height=15
        )

        self.history_tree.heading("timestamp", text="Время")
        self.history_tree.heading("operation", text="Операция")
        self.history_tree.heading("node_id", text="Node ID")
        self.history_tree.heading("value", text="Значение")
        scroll_y = ttk.Scrollbar(frame, orient="vertical", command=self.history_tree.yview)
        scroll_x = ttk.Scrollbar(frame, orient="horizontal", command=self.history_tree.xview)
        self.history_tree.configure(yscrollcommand=scroll_y.set, xscrollcommand=scroll_x.set)

        self.history_tree.grid(row=0, column=0, sticky="nsew")
        scroll_y.grid(row=0, column=1, sticky="ns")
        scroll_x.grid(row=1, column=0, sticky="ew")

        # Панель фильтрации
        filter_frame = ttk.Frame(frame)
        filter_frame.grid(row=2, column=0, columnspan=2, sticky="ew", pady=5)

        ttk.Label(filter_frame, text="Фильтр:").pack(side=tk.LEFT)
        self.history_filter = ttk.Entry(filter_frame)
        self.history_filter.pack(side=tk.LEFT, expand=True, fill=tk.X, padx=5)

        ttk.Button(
            filter_frame,
            text="Применить",
            command=self._apply_history_filter
        ).pack(side=tk.LEFT)

        ttk.Button(
            filter_frame,
            text="Сбросить",
            command=self._reset_history_filter
        ).pack(side=tk.LEFT)

        ttk.Button(
            filter_frame,
            text="Обновить",
            command=self._refresh_history
        ).pack(side=tk.RIGHT)

        frame.columnconfigure(0, weight=1)
        frame.rowconfigure(0, weight=1)

        self._refresh_history()

    def _refresh_history(self):
        """Обновление истории операций"""
        if not hasattr(self, 'history_tree'):
```

Продолжение листинга В.1

```
            return

        try:
            self.history_tree.delete(*self.history_tree.get_children())
            history = self.db.get_node_operations_history(limit=100) if ha-
sattr(self, 'db') else []

            for item in history:
                self.history_tree.insert("", "end", values=(
                    item.get('timestamp', ''),
                    item.get('operation_type', ''),
                    item.get('node_id', ''),
                    str(item.get('value', ''))[:100]
                ))
        except Exception as e:
            logging.error(f"Ошибка обновления истории: {str(e)}")
self._update_status(f"Ошибка загрузки истории: {str(e)}")

    def _setup_logs_tab(self):
        """Настройка вкладки логов"""
        frame = ttk.Frame(self.logs_tab)
        frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        # Текстовое поле для логов
        self.log_text = tk.Text(
            frame,
            wrap=tk.WORD,
            state=tk.DISABLED,
            font=('Courier New', 10)
        )

        scrollbar = ttk.Scrollbar(frame, command=self.log_text.yview)
        self.log_text.configure(yscrollcommand=scrollbar.set)

        self.log_text.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
        scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        # Кнопка очистки логов
        ttk.Button(
            frame,
            text="Очистить логи",
            command=self._clear_logs
        ).pack(side=tk.BOTTOM, pady=5)

        self._setup_log_handler()

    def _setup_log_handler(self):
        """Настройка обработчика логов для вывода в интерфейс"""
        class TextHandler(logging.Handler):
            def __init__(self, text_widget):
                super().__init__()
                self.text_widget = text_widget

            def emit(self, record):
                msg = self.format(record)
                self.text_widget.config(state=tk.NORMAL)
                self.text_widget.insert(tk.END, msg + "\n")
                self.text_widget.config(state=tk.DISABLED)
                self.text_widget.see(tk.END)
```

Продолжение листинга В.1

```
        handler = TextHandler(self.log_text)
        handler.setFormatter(logging.Formatter('%(asctime)s  -  %(levelname)s  -
%(message)s'))
        logging.getLogger().addHandler(handler)

    def _clear_logs(self):
        """Очистка логов"""
        self.log_text.config(state=tk.NORMAL)
        self.log_text.delete(1.0, tk.END)
        self.log_text.config(state=tk.DISABLED)
    def _refresh_data(self):
        """Обновление всех данных"""
        pass

    def _refresh_monitor_data(self):
        """Обновление данных мониторинга"""
        try:
            if hasattr(self, 'subscriptions_tree'):
                self.subscriptions_tree.delete(*self.subscrip-
tions_tree.get_children())
                subs = self.db.get_active_subscriptions()
                for sub in subs:
                    self.subscriptions_tree.insert("", "end", values=(
                        sub['node_id'],
                        "Активна",
                        sub.get('last_value', 'N/A')
                    ))

            if hasattr(self, 'events_tree'):
                self.events_tree.delete(*self.events_tree.get_children())
                events = self.db.get_events(limit=50)
                for event in events:
                    self.events_tree.insert("", "end", values=(
                        event['timestamp'].strftime("%Y-%m-%d %H:%M:%S"),
                        event['event_type'],

                        event['message']
                    ))

        except Exception as e:
            logging.error(f"Ошибка обновления мониторинга: {str(e)}")
            self._update_status(f"Ошибка обновления: {str(e)}")

    def _update_tag_info(self, node_id: str):
        """Обновление информации о теге"""
        if not hasattr(self, 'tag_info_text'):
            return

        try:
            self.tag_info_text.config(state=tk.NORMAL)
            self.tag_info_text.delete(1.0, tk.END)
            # Получение информации о теге из базы
            tag_info = self.db.get_tag_info(node_id) if hasattr(self, 'db') else
None
            current_value = "Не доступно"

            # Получение текущего значения из OPC сервера
            if hasattr(self, 'opcua_client') and self.opcua_client.client:
                try:
```

Продолжение листинга В.1

```
                    current_value              =              str(self.opcua_cli-
ent.safe_read_node(node_id))
                except:
                    current_value = "Ошибка чтения"

            info_text   =   f"Node   ID:   {node_id}\nТекущее   значение:   {cur-
rent_value}\n"
            if tag_info:

                info_text  +=  f"Тип  данных:  {tag_info.get('data_type',  'un-
known')}\n"
                info_text += f"Описание: {tag_info.get('description', 'нет')}\n"
                info_text += f"ID в БД: {tag_info.get('id', 'N/A')}"
            else:
                info_text += "(Тег не зарегистрирован в базе данных)"

            self.tag_info_text.insert(tk.END, info_text)

        except Exception as e:
            logging.error(f"Ошибка обновления информации: {str(e)}")
            self.tag_info_text.insert(tk.END, f"Ошибка: {str(e)}")
        finally:
            self.tag_info_text.config(state=tk.DISABLED)

    def _connect(self):
        """Подключение к OPC UA серверу"""
        endpoint = self.endpoint_entry.get()

        def connect_thread():
            try:
                self.root.after(0, lambda: (

                    self.connect_btn.config(state=tk.DISABLED),
                    self.root.config(cursor="watch")
                ))

                # Подключение к серверу
                self.opcua_client.connect(endpoint)

                self.root.after(0, lambda: (
                    self.disconnect_btn.config(state=tk.NORMAL),
                    self.read_btn.config(state=tk.NORMAL),
                    self.write_btn.config(state=tk.NORMAL),
                    self.subscribe_btn.config(state=tk.NORMAL),
                    self._refresh_events(),
                    self._refresh_subscriptions(),
                    self.root.config(cursor="")
                ))

            except Exception as ex:
                error = str(ex)
                self.root.after(0, lambda: (
                    messagebox.showerror("Ошибка",  f"Не  удалось  подключиться:
{error}"),
                    self.connect_btn.config(state=tk.NORMAL),
                    self.root.config(cursor="")
                ))
```

Продолжение листинга В.1

```
        # Запуск в отдельном потоке
        threading.Thread(target=connect_thread, daemon=True).start()

    def _disconnect(self):
        """Отключение от OPC UA сервера"""
        def disconnect_thread():
            try:
                # Блокируем кнопки во время отключения
                self.root.after(0, lambda: (
                    self.disconnect_btn.config(state=tk.DISABLED),
                    self.root.config(cursor="watch")
                ))

                self.opcua_client.disconnect()

                self.root.after(0, lambda: (
                    self.connect_btn.config(state=tk.NORMAL),
                    self.disconnect_btn.config(state=tk.DISABLED),
                    self.read_btn.config(state=tk.DISABLED),
                    self.write_btn.config(state=tk.DISABLED),
                    self.subscribe_btn.config(state=tk.DISABLED),
                    self._refresh_events(),
                    self._refresh_subscriptions(),
                    self.root.config(cursor="")
                ))

            except Exception as ex:
                error = str(ex)
                self.root.after(0, lambda: (
                    messagebox.showerror("Ошибка", f"Не удалось отключиться: {er-
ror}"),
                    self.disconnect_btn.config(state=tk.NORMAL),
                    self.root.config(cursor="")
                ))

        threading.Thread(target=disconnect_thread, daemon=True).start()

    def _read_node(self):
        """Чтение значения узла"""
        node_id = self.node_id_entry.get().strip()

        if not node_id:
            self._update_status("Ошибка: NodeID не указан")
            messagebox.showerror("Ошибка", "Введите NodeID для чтения")
            return

        self._set_ui_state(False)
        self.progress_bar.start()
        self._update_status(f"Чтение узла {node_id}...")

        def read_worker():
            try:
                if not self.opcua_client.node_exists(node_id):
                    self.root.after(0, lambda: [
                        messagebox.showerror("Ошибка",   f"Узел   {node_id}   не
найден"),
                        self._update_status(f"Узел {node_id} не найден")
                    ])
                    Return
```

Продолжение листинга В.1

```
            value = self.opcua_client.safe_read_node(node_id)

            self.root.after(0, lambda: [
                self.value_entry.delete(0, tk.END),
                self.value_entry.insert(0, str(value)),
                self._update_tag_info(node_id),
                self._update_status(f"Узел {node_id} прочитан успешно")
            ])

        except Exception as ex:
            error = str(ex)
            self.root.after(0, lambda: [
                messagebox.showerror("Ошибка", f"Ошибка чтения: {error}"),
                self._update_status(f"Ошибка чтения {node_id}")
            ])
        finally:
            self.root.after(0, lambda: [
                self.progress_bar.stop(),

                self._set_ui_state(True)
            ])

    threading.Thread(target=read_worker, daemon=True).start()

def _set_ui_state(self, enabled: bool):
    """Установка состояния элементов интерфейса"""
    state = tk.NORMAL if enabled else tk.DISABLED

    widgets = [
        'connect_btn', 'disconnect_btn', 'refresh_btn',
        'read_btn', 'write_btn', 'subscribe_btn',
        'endpoint_entry', 'username_entry', 'password_entry',
        'node_id_entry', 'value_entry'
    ]

    for widget_name in widgets:
        if hasattr(self, widget_name):
            widget = getattr(self, widget_name)
            widget.config(state=state)

def _update_status(self, message: str):
    """Обновление строки состояния"""
    if hasattr(self, 'status_var'):
        self.status_var.set(message)
    if hasattr(self, 'status_bar'):
        self.status_bar.update()

def _refresh_all(self):
    """Полное обновление всех данных"""
    try:
        self._update_status("Обновление данных...")
        self.progress_bar.start()
        self._set_ui_state(False)
        self._refresh_connection_info()
        self._refresh_node_info()
        self._refresh_subscriptions()
        self._refresh_events()
        self._refresh_node_history()
        self._refresh_logs()
```

Продолжение листинга В.1

```
            self._update_status("Данные обновлены")
        except Exception as e:
            self._update_status(f"Ошибка обновления: {str(e)}")
            logging.error(f"Ошибка при обновлении данных: {str(e)}")
        finally:
            self.progress_bar.stop()
            self._set_ui_state(True)

    def _write_node(self):
        """Запись значения в узел"""
        node_id = self.node_id_entry.get().strip()
        value = self.value_entry.get().strip()

        if not node_id:
            self._update_status("Ошибка: не указан NodeID")
            messagebox.showerror("Ошибка", "Введите NodeID для записи")
            return

        if not value:
            self._update_status("Ошибка: не указано значение")
            messagebox.showerror("Ошибка", "Введите значение для записи")
            return

        try:
            # Проверка прав на запись
            if not self.opcua_client.check_write_permission(node_id):
                self._update_status(f"Нет прав на запись в {node_id}")
                messagebox.showerror(
                    "Ошибка прав",
                    f"Нет прав на запись в узел {node_id}\n\n"
                    "Возможные причины:\n"
                    "1. Неправильные учетные данные\n"
                    "2. Недостаточные права пользователя\n"
                    "3. Узел доступен только для чтения\n\n"
                    "Проверьте настройки сервера OPC UA"
                )
                return
        except Exception as e:
            error_msg = f"Ошибка проверки прав: {str(e)}"
            logging.error(error_msg)
            self._update_status(error_msg)
            messagebox.showerror("Ошибка", error_msg)
            return

        self._set_ui_state(False)
        self.progress_bar.start()
        self._update_status(f"Запись {value} в {node_id}...")

        def write_worker():
            try:
                self.opcua_client.write_node(node_id, value)

                self.root.after(0, lambda: [
                    self._update_status(f"Значение записано в {node_id}"),
                    self._update_tag_info(node_id),
                    messagebox.showinfo("Успех", "Значение успешно записано")
                ])

            except Exception as e:
```

Продолжение листинга В.1

```
            error_msg = f"Ошибка записи в {node_id}: {str(e)}"

            logging.error(error_msg, exc_info=True)
            self.root.after(0, lambda: [
                self._update_status(error_msg),
                messagebox.showerror("Ошибка", error_msg)
            ])

        finally:
            self.root.after(0, lambda: [
                self.progress_bar.stop(),
                self._set_ui_state(True)
            ])

    threading.Thread(target=write_worker, daemon=True).start()

def _subscribe_node(self):
    """Подписка на изменения узла"""
    node_id = self.node_id_entry.get().strip()
    if not node_id:
        messagebox.showerror("Ошибка", "Введите NodeID для подписки")
        return

    self._set_ui_state(False)
    self.progress_bar.start()
    self._update_status(f"Попытка подписки на {node_id}...")

    def callback(nid, val):
        """Коллбэк при изменении значения узла"""
        self.root.after(0, lambda: [
            self.value_entry.delete(0, tk.END),
            self.value_entry.insert(0, str(val)),
            self._update_tag_info(nid)
        ])

    def worker():
        try:
            if not self.opcua_client.node_exists(node_id):
                raise ValueError(f"Узел {node_id} не существует")

            self.opcua_client.subscribe(node_id, callback)
            self.root.after(0, lambda: [
                self._update_status(f"Подписка на {node_id} активна"),
                messagebox.showinfo("Успех", "Подписка создана"),
                self._refresh_subscriptions(),
                self._refresh_events()
            ])
        except ValueError as e:
            self.root.after(0, lambda: [
                self._update_status(f"Ошибка: {str(e)}"),
                messagebox.showerror("Ошибка подписки", str(e))
            ])

        except Exception as e:
            error_msg = f"Неожиданная ошибка: {str(e)}"
            logging.error(error_msg)
            self.root.after(0, lambda: [
                self._update_status(error_msg),
                messagebox.showerror("Ошибка", error_msg)
```

Продолжение листинга В.1

```
                ])
            finally:
                self.root.after(0, lambda: [
                    self.progress_bar.stop(),
                    self._set_ui_state(True)
                ])

        threading.Thread(target=worker, daemon=True).start()

    def _unsubscribe(self):
        """Отписка от изменений узла"""
        selected = self.subscriptions_tree.selection()
        if not selected:
            messagebox.showwarning("Предупреждение", "Не выбрана подписка для от-
писки")
            return

        item = self.subscriptions_tree.item(selected[0])
        node_id = item['values'][0]

        if not messagebox.askyesno(
                "Подтверждение",
                f"Вы уверены, что хотите отписаться от узла {node_id}?"
        ):
            return

        try:
            self.opcua_client.unsubscribe(node_id)

            self.subscriptions_tree.delete(selected[0])
            self._update_status(f"Успешно отписались от узла {node_id}")

            self._refresh_events()

        except Exception as e:
            error_msg = f"Ошибка отписки: {str(e)}"
            logging.error(error_msg)
            messagebox.showerror("Ошибка", error_msg)

    def _refresh_events(self):
        """Обновление списка событий"""
        try:
            self.events_tree.delete(*self.events_tree.get_children())

            events = self.db.get_events(limit=100)

            for event in events:
                self.events_tree.insert("", tk.END, values=(
                    event.get('timestamp', '').strftime("%Y-%m-%d %H:%M:%S"),
                    event.get('event_type', ''),
                    event.get('tag_name', '') or event.get('node_id', ''),
                    event.get('message', ''),
                    event.get('severity', '')
                ))

        except Exception as e:
            error_msg = f"Ошибка обновления событий: {str(e)}"
            logging.error(error_msg)
            self._update_status(error_msg)
```

Окончание листинга В.1

```
class TextHandler(logging.Handler):
    """Обработчик логов для вывода в текстовое поле"""
    def __init__(self, text_widget):
        super().__init__()
        self.text_widget = text_widget
        self.setFormatter(logging.Formatter('%(asctime)s - %(levelname)s - %(mes-
sage)s'))

    def emit(self, record):
        """Вывод сообщения в текстовое поле"""
        msg = self.format(record)
        self.text_widget.config(state=tk.NORMAL)

        self.text_widget.insert(tk.END, msg + "\n")
        self.text_widget.config(state=tk.DISABLED)
        self.text_widget.see(tk.END)
```

ПРИЛОЖЕНИЕ Г
ИНТЕГРИРОВАННЫЕ МОДЕЛИ И СЕРВИСЫ OPC UA

ИНТЕГРИРОВАННАЯ МОДЕЛЬ АДРЕСНОГО ПРОСТРАНСТВА OPC UA

Набор объектов и связанной с ними информации, которые сервер OPC UA предоставляет клиентам, называется его адресным пространством. Адресное пространство представляет его содержимое в виде набора узлов, соединенных ссылками

Основные характеристики узлов описываются атрибутами, определяемыми OPC. Атрибуты – это единственные элементы сервера, которые имеют значения данных. Типы данных, определяющие значения атрибутов, могут быть простыми или сложными.

Узлы в адресной области типизируются в соответствии с их назначением и значением. Классы узлов определяют метаданные для адресной области OPC UA. Модель узла представлена на рисунке Г.1[19].

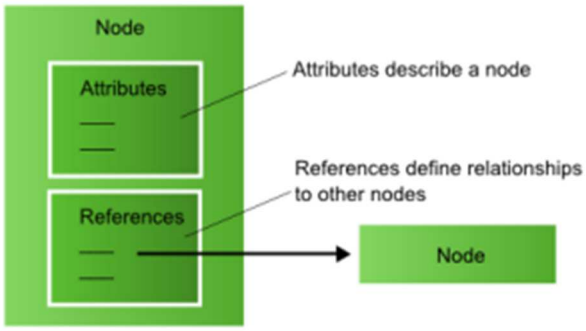

Рисунок Г.1 – Модель узла

Базовый класс узлов определяет атрибуты, общие для всех узлов, позволяющие идентифицировать, классифицировать и именовать их. Каждый класс узлов наследует эти атрибуты и может дополнительно определять свои собственные атрибуты.

В спецификации OPC UA определено восемь различных классов узлов:

1. объект. Данный класс используется для отображения систем, их компонентов, реальных объектов и программного обеспечения.

2. переменная. Данный класс служит для отображения содержания объектов.

3. метод. Данный класс необходим для отображения различных методов в адресном пространстве.

4. класс «Отображение» предназначен для указания видимости узлов и отношений между ними в адресном пространстве.

5. класс «Тип объекта» описывает типы узлов объектов в адресном пространстве сервера.

6. тип переменной. Описывает тип переменной, хранимой в адресном пространстве.

7. тип отношения. Данный класс описывает типы отношений, используемых сервером.

8. тип данных. Данный тип отображает абстрактность данных. Все типы данных представлены как узлы классов узлов в адресном пространстве. Имеет один атрибут – значения абстрактно или нет.

Каждый узел в адресном пространстве является экземпляром одного из классов узлов. Клиентам и серверам не разрешается определять дополнительные классы узлов или расширять определения этих классов узлов (например, список атрибутов для класса узла).

Ссылки используются для установления связей между узлами, а доступ к ним осуществляется посредством служб просмотра и запросов. Как и атрибуты, ссылки являются ключевым компонентом узлов. Однако в отличие от атрибутов, ссылки определяются как экземпляры класса «тип отношений», который виден в адресном пространстве и определяется с помощью этого же класса.

Узел, содержащий ссылку, называется исходным узлом, тогда как узел, на который делается ссылка – целевым. Комбинация исходного узла, типа отношений и целевого узла используется для уникальной идентификации ссылок в службах OPC UA. Это означает, что каждый узел может иметь только одну ссылку на другой узел с тем же типом отношений.

Целевой узел может находиться как в адресном пространстве исходного сервера OPC UA, так и на другом сервере. В случае если целевой узел находится на удаленном сервере, его идентификация происходит с помощью комбинации имени удаленного сервера и присвоенного ему идентификатора.

Переменные используются для отображения значений данных и делятся на два основных типа: свойства и переменные данных. Эти типы различаются по представляемым данным и возможности содержать другие переменные.

Свойства содержат метаданные объектов или других элементов системы OPC UA. Они могут быть добавлены пользователем самостоятельно для описания характеристик объектов или других элементов системы (например, устройства или порядок опроса устройств). Свойства отличаются от атрибутов тем, что могут быть добавлены динамически.

Например: Атрибут может определить тип данных переменной (например, float), тогда как свойство может использоваться для указания единиц измерения этой переменной (например, градусы Цельсия). Чтобы избежать рекурсии при использовании свойств запрещено использовать свойства внутри самих себя. Для легкой идентификации имя каждого свойства должно быть уникальным внутри контекста своего родительского узла. Узлы со своими свойствами должны находиться на одном сервере OPC UA.

Переменные данных предназначены для определения содержимого объектов – контейнеров методов или других элементов системы OPC UA.

В отличие от объектных узлов, которые не имеют собственных значений данные из таких контейнеров можно получить через специальные узлы-переменные. Например: Файл использует данные переменных чтобы представить свое содержимое в виде массива байтов; его свойства можно использовать чтобы указать время создания файла или имя автора файла.

Также серверы могут разделять адресное пространство на представления тем самым упрощая доступ клиентов.

ИНТЕГРИРОВАННАЯ ОБЪЕКТНАЯ МОДЕЛЬ

Объектная модель OPC UA предоставляет согласованный интегрированный набор классов узлов для представления объектов в адресном пространстве. Эта модель представляет объекты в терминах их переменных, событий и методов, а также их связей с другими объектами.

Объектная модель OPC UA позволяет серверам предоставлять определения типов для объектов и их компонентов. Определения типов могут быть разделены на подклассы. Они также могут быть общими или зависеть от конкретной системы. Типы объектов могут быть определены организациями по стандартизации, поставщиками или конечными пользователями.

Эта модель позволяет интегрировать данные, сигналы тревоги и события, а также их историю в единый OPC UA сервер. Например, серверы OPC UA могут представлять датчик температуры в виде объекта, который состоит из значения температуры, набора параметров тревоги и соответствующего набора пределов тревоги

Использование объектно-ориентированного подхода в OPC UA не является обязательным требованием. Аналогично OPC Data Access, простую модель адресного пространства можно реализовать с помощью объектов, папок и переменных. Однако наличие дополнительных функций, характерных для объектно-ориентированного подхода, существенно облегчает разработку систем на основе спецификации OPC UA.

Эти дополнительные функции позволяют более эффективно организовать структуру данных и взаимодействие между компонентами системы. Это особенно полезно при работе с сложными системами управления и мониторинга, где требуется гибкость и масштабируемость.

Объектно-ориентированный подход в OPC UA обеспечивает возможность создавать модульные решения за счет использования таких концепций как наследование, полиморфизм и инкапсуляция. Это позволяет разработчикам проектировать системы управления данными более прозрачно для пользователя. На рисунке Г.2 [19] представлена модель объекта OPC UA сервера.

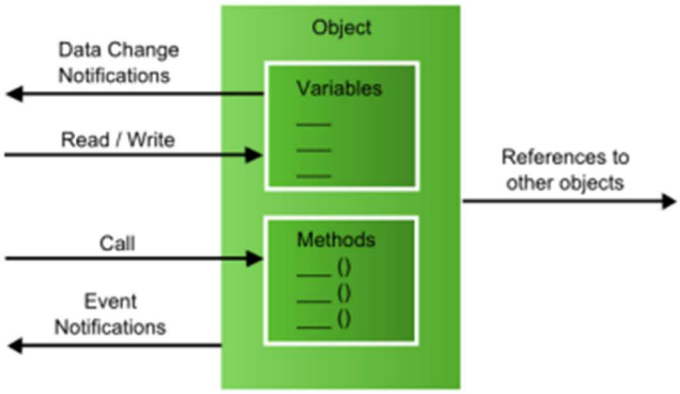

Рисунок Г.2 – Модель объекта OPC UA сервера

Службы UA предназначены для взаимодействия с объектами и их компонентами. Они позволяют выполнять операции чтения и записи значений переменных, вызов методов объектов и получение событий от них (например, уведомления об изменении значений). Для анализа связей между объектами и их компонентами используются службы просмотра.

Эти службы выступают в качестве клиента сервера OPC UA.В адресном пространстве модели элементами являются узлы. Каждый узел имеет определенный класс – например, может быть объектом, переменной или методом объекта – и представляет собой отдельный элемент модели объекта.

ИНТЕГРИРОВАННЫЕ СЕРВИСЫ

Интерфейс между клиентами и серверами OPC UA определяется как набор сервисов. Эти сервисы организованы в логические группы, называемые наборами сервисов.

Сервисы OPC UA предоставляют клиентам две возможности. Они позволяют клиентам отправлять запросы на серверы и получать от них ответы. Они также позволяют клиентам подписываться на серверы для получения уведомлений.

Уведомления используются сервером для сообщения о таких событиях, как аварийные сигналы, изменения значений данных, события и результаты выполнения программы.

Сообщения OPC UA могут быть закодированы в виде текста XML или в двоичном формате для повышения эффективности. Они могут передаваться с использованием нескольких базовых транспортных средств, например, TCP или веб-служб по протоколу HTTP.