Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационных компьютерных систем

Дисциплина "Объектно-ориентированное программирование"

*К защите допустить*:
Руководитель курсовой работы
старший преподаватель
кафедры
_____ А.В.Михалькевич
22.01.2025

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

на тему

# Программа отправки и получения текстовых сообщений

БГУИР КР 1-40 05 01-10 № 148 ПЗ

Студент (подпись студента)

Курсовая работа
представлена на проверку
22.01.2025
_____
(подпись студента)

2025

# Реферат

БГУИР КР 1-40 05 01-10 № 148 ПЗ, гр. 814303

, Программа отправки и получения текстовых сообщений, Минск: БГУИР - 2025.

Пояснительная записка 129394 с., 1 рис., 0 табл.

Ключевые слова: MQTT-клиент, Java

Предмет Объектно-ориентированное программирование, А.В.Михалькевич

*Предмет: создание программы MQTT-клиента на языке программирования Java. Объект: шаблоны проектирования, разработка пользовательского интерфейса. Цель: создание и проектирование графического пользовательского интерфейса для обмена текстовыми сообщениями между клиентами сети Ethernet по протоколу MQTT. Ссылка на онлайн-репозиторий GitHub: https://github.com/ilyazenkovsky/zenkovsky_kursach_oop*

*Subject: creating an MQTT client program in the Java programming language. Object: design templates, user interface development. Goal: create and design a graphical user interface for text messaging between Ethernet clients over the MQTT Protocol. Link to the github online repository: https://github.com/ilyazenkovsky/zenkovsky_kursach_oop*

# Содержание



# Введение

С развитием промышленности увеличивается количество устройств, которые нужно контролировать и получать от них различные данные. Для решения проблем взаимодействия большого количества устройств и проблем объединения устройств в одну сеть была создана концепция Интернета вещей (англ. Internet of Things, IoT) – это когда устройства объединяются по какому-то признаку в одну сеть, потом уже несколько подобных сетей объединяются в другую большую сеть и так далее.Устройства в таких сетях взаимодействуют друг с другом по средствам различных интерфейсов и протоколов передачи данных. Так как мы говорим о

промышленном применении концепции IoT, в которой должны использоваться промышленное оборудование со своими протоколами и аппаратными средствами, то мы переходим к концепции IIoT (Промышленного Интернета вещей). Протокол MQTT, на текущее время, завоевал свою популярность и стал стандартом де факто в проектах, направленных на создание решений для IoT (Интернета вещей). MQTT или Message Queue Telemetry Transport — упрощённый сетевой протокол, работающий поверх TCP/IP. Используется для обмена сообщениями между устройствами по принципу издатель-подписчик. Первая версия протокола была разработана доктором Энди Станфорд-Кларком (IBM) и Арлен Ниппер (Arcom) в 1999 году и опубликована под роялти-фри лицензией. Спецификация MQTT 3.1.1 была стандартизирована консорциумом OASIS в 2014 году. Это легкий, компактный и открытый протокол обмена данными созданный для передачи данных на удалённых локациях, где требуется небольшой размер кода и есть ограничения по пропускной способности канала. Конечно, Интернет вещей – это определенная абстракция или, правильнее, концепция построения распределенных сетей устройств или машин. Актуальность темы данной курсовой работы заключается в эффективном рещении частных задач взаимодействия машин Machine-to-Machine (M2M), включая и уровень подключения этих систем к Интернет, как для создания промышленных систем автоматики, так, например, и для построения систем «умного дома». Очевидно, что для критически важных систем требуется наличие локального арбитра или брокера и устройств, позволяющих отработать решение ситуации не зависимо от качества Интернет-подключения, а также в случае полного разрыва связи. При этом, в качестве аппаратной платформы для запуска такого брокера MQTT, отлично себя зарекомендовали энергоэффективные, но от этого не менее производительные, микропроцессоры на базе архитектуры ARM. Такая взаимосвязь аппаратной платформы, протоколов обмена сообщениями и их программной реализации стала отправной точкой в стремительном прогрессе развития современных систем автоматизации. Также существует версия протокола MQTT-SN (MQTT for Sensor Networks), ранее известная как MQTT-S, которая предназначена для встраиваемых беспроводных устройств без поддержки TCP/IP сетей, например, Zigbee. Таким образом, целью данного курсового проекта является создание и проектирование графического пользовательского интерфейса для обмена текстовыми сообщениями между клиентами сети Ethernet по протоколу MQTT. Задачей данной курсовой работы является выполнение следующих этапов: — Создание графического интерфейса для отправки текстовых сообщений; — Реализация протокола MQTT с использованием ООП.

# 1 МАТЕМАТИЧЕСКАЯ МОДЕЛЬ СЕТИ ETHERNET КАК СИСТЕМЫ МАССОВОГО ОБСЛУЖИВАНИЯ

### 1.1. Понятие системы массового обслуживания (СМО)

Система массового обслуживания (СМО) — система, которая производит обслуживание поступающих в неё требований. Обслуживание требований в СМО осуществляется обслуживающими приборами. Классическая СМО содержит от одного до бесконечного числа приборов. В зависимости от наличия возможности ожидания поступающими требованиями начала обслуживания СМО подразделяются на:

- системы с потерями, в которых требования, не нашедшие в момент поступления ни одного свободного прибора, теряются;

- системы с ожиданием, в которых имеется накопитель бесконечной ёмкости для буферизации поступивших требований, при этом ожидающие требования образуют очередь;

- системы с накопителем конечной ёмкости (ожиданием и ограничениями), в которых длина

очереди не может превышать ёмкости накопителя; при этом требование, поступающее в переполненную СМО (отсутствуют свободные места для ожидания), теряются.

Выбор требования из очереди на обслуживание производится с помощью так называемой дисциплины обслуживания. Их примерами являются *FCFS/FIFO* (пришедший первым обслуживается первым), *LCFS/LIFO* (пришедший последним обслуживается первым), *random* (случайный выбор). В системах с ожиданием накопитель в общем случае может иметь сложную структуру.

Поступив в обслуживающую систему, требование присоединяется к очереди других (ранее поступивших) требований. Канал обслуживания выбирает требование из находящихся в очереди, с тем, чтобы приступить к его обслуживанию. После завершения процедуры обслуживания очередного требования канал обслуживания приступает к обслуживанию следующего требования, если таковое имеется в блоке ожидания.

Цикл функционирования системы массового обслуживания подобного рода повторяется многократно в течение всего периода работы обслуживающей системы. При этом предполагается, что переход системы на обслуживание очередного требования после завершения обслуживания предыдущего требования происходит мгновенно, в случайные моменты времени.

Примерами систем массового обслуживания могут служить:

- посты технического обслуживания автомобилей;

- посты ремонта автомобилей;

- персональные компьютеры, обслуживающие поступающие заявки или требования на решение тех или иных задач;

- станции технического обслуживания автомобилей;

- аудиторские фирмы;

- отделы налоговых инспекций, занимающиеся приемкой и проверкой текущей отчетности предприятий;

- телефонные станции и т. д.

Основными компонентами системы массового обслуживания любого вида являются:

- входной поток поступающих требований или заявок на обслуживание;

- дисциплина очереди;

- механизм обслуживания.

Все системы массового обслуживания различают по числу каналов обслуживания на:

- одноканальные системы;

- многоканальные системы.

**1.2. Задачи одноканальной СМО с ограниченной очередью**

В СМО с ограниченной очередью число мест $m$ в очереди ограничено. Следовательно, заявка, поступившая в момент времени, когда все места в очереди заняты, отклоняется и покидает СМО. Если заявка застаёт канал свободным, то она принимается на обслуживание и обслуживается каналом. После окончания обслуживания канал освобождается. Дисциплина очереди естественная: кто раньше пришёл, тот раньше и обслуживается. Максимальное число мест в очереди m.

Граф такой СМО представлен на рисунке 1.2.

***Рисунок 1.2 – Граф одноканальной СМО с ограниченной очередью в файле "Формулы и диаграмы"***

Состояния СМО представляются следующим образом:

- $S_0$ - канал обслуживания свободен;

- $S_1$ – канал обслуживания занят, но очереди нет;

- $S_2$ – канал обслуживания занят, в очереди одна заявка;

- $S_{k+1}$ – канал обслуживания занят, в очереди $k$ заявок;

- $S_{m+1}$ – канал обслуживания занят, все $m$ мест в очереди заняты.

Будем предполагать, что входящий поток заявок на обслуживание есть простейший поток с интенсивностью $\lambda$.

Интенсивность потока обслуживания равна $\mu$. Длительность обслуживания – случайная величина, подчиненная показательному закону распределения. Поток обслуживаний является простейшим пуассоновским потоком событий.

Система уравнений, описывающих процесс в этой системе, имеет решение:

***Формула 1.1. в файле "Формулы и диаграмы"***

Знаменатель первого выражения представляет собой геометрическую прогрессию с первым членом 1 и знаменателем $\rho$, откуда получаем

***Формула 1.2. в файле "Формулы и диаграмы"***

***Формула 1.3. в файле "Формулы и диаграмы"***

и предельные вероятности приобретают вид:

***Формула 1.4. в файле "Формулы и диаграмы"***

***Формула 1.5. в файле "Формулы и диаграмы"***

***Формула 1.6. в файле "Формулы и диаграмы"***

***Формула 1.7. в файле "Формулы и диаграмы"***

Выполнение условия стационарности $\rho < 1$ необязательно, поскольку число заявок в СМО контролируется путем введения ограничения на длину очереди. Однако выражение справедливо только при $\rho < 1$ (поскольку для $\rho = 1$ получается неопределенность вида 0/0). Сумма геометрической прогрессии со знаменателем $\rho = 1$ равна в этом случае $m + 2$ и

Определим характеристики одноканальной СМО с ожиданием и ограниченной длиной очереди, равной $m$:

Вероятность отказа в обслуживании заявки (отказ произойдет в случае, если канал занят и в очереди находятся $m$ заявок):

***Формула 1.8. в файле "Формулы и диаграмы"***

Относительная пропускная способность.

***Формула 1.9. в файле "Формулы и диаграмы"***

Абсолютная пропускная способность.

***Формула 1.10. в файле "Формулы и диаграмы"***

Среднее число находящихся в очереди заявок.

В случае, когда $\rho$ отлично от 1, можно воспользоваться формулой

***Формула 1.11. в файле "Формулы и диаграмы"***

При $\rho = 1$ можно прибегнуть к прямому подсчету

***Формула 1.12. в файле "Формулы и диаграмы"***

**Среднее число находящихся в системе заявок.**

Поскольку среднее число находящихся в системе заявок

***Формула 1.13. в файле "Формулы и диаграмы"***

где - среднее число заявок, находящихся под обслуживанием, то зная остается найти . Т.к. канал один, то число обслуживаемых заявок может равняться либо 0, либо 1 с вероятностями *P0* и *P1=1- P0* соответственно, откуда

***Формула 1.14. в файле "Формулы и диаграмы"***

и среднее число находящихся в системе заявок равно

***Формула 1.15. в файле "Формулы и диаграмы"***

Среднее время ожидания заявки в очереди.

***Формула 1.16. в файле "Формулы и диаграмы"***

То есть, среднее время ожидания заявки в очереди равно среднему числу заявок в очереди, деленному на интенсивность потока заявок.

**Среднее время пребывания заявки в системе.**

Время пребывания заявки в системе складывается из времени ожидания заявки в очереди и времени обслуживания . Если загрузка системы составляет 100%, то $=1/\mu$, в противном случае $=q/\mu$. Отсюда

***Формула 1.17. в файле "Формулы и диаграмы"***

## 1.3. Задачи многоканальной СМО с ограниченными очередями

Рассмотрим $n$-канальную систему массового обслуживания с ожиданием.

Будем считать входящий поток заявок на обслуживание простейшим потоком с интенсивностью $\lambda$.

Интенсивность потока обслуживания равна $\mu$. Длительность обслуживания – случайная величина, подчиненная показательному закону распределения. Поток обслуживаний является простейшим пуассоновским потоком событий.

Размер очереди допускает нахождение в ней $m$ заявок.

Для нахождения предельных вероятностей можно использовать следующие выражения.

***Формула 1.18. в файле "Формулы и диаграмы"***

***Формула 1.19. в файле "Формулы и диаграмы"***

***Формула 1.20. в файле "Формулы и диаграмы"***

Вероятность отказа в обслуживании заявки (отказ произойдет в случае, если все каналы заняты и в очереди находятся $m$ заявок):

***Формула 1.21. в файле "Формулы и диаграмы"***

Относительная пропускная способность.

***Формула 1.22. в файле "Формулы и диаграмы"***

Абсолютная пропускная способность.

***Формула 1.23. в файле "Формулы и диаграмы"***

Среднее число занятых каналов.

Для СМО с очередью среднее число занятых каналов не совпадает (в отличие от СМО с отказами) со средним числом заявок в системе. Отличие равно числу заявок, ожидающих в очереди.

Обозначим среднее число занятых каналов . Каждый занятый канал обслуживает в среднем $\mu$ заявок в единицу времени, а СМО в целом – А заявок в единицу времени. Разделив А на $\mu$ получим

***Формула 1.24. в файле "Формулы и диаграмы"***

Среднее число находящихся в очереди заявок.

Для нахождения среднего числа ожидающих в очереди заявок в случае, если $\chi \neq 1$, можно использовать выражение:

***Формула 1.25. в файле "Формулы и диаграмы"***

Для $\chi = 1$ необходимо подсчитать сумму:

***Формула 1.26. в файле "Формулы и диаграмы"***

Среднее число находящихся в системе заявок.

***Формула 1.27. в файле "Формулы и диаграмы"***

Среднее время ожидания заявки в очереди.

Среднее время ожидания заявки в очереди можно найти из выражения ($\chi \neq 1$).

***Формула 1.28. в файле "Формулы и диаграмы"***

Среднее время пребывания заявки в системе.

Так же как и в случае с одноканальной СМО имеем:

***Формула 1.29. в файле "Формулы и диаграмы"***

# 2 ETHERNET-ПРОТОКОЛЫ

## 2.1 Протоколы UDP, TCP , DHCP

Локальная сеть *Ethernet* (англ. *ether* — «эфир» и *network* — «сеть, цепь») — семейство технологий пакетной передачи данных между устройствами для компьютерных и промышленных сетей, которая поддерживает неявный обмен сообщениями (обмен сообщениями ввода/вывода в реальном времени), явный обмен (обмен сообщениями) или оба и использует широко распространённые коммерческие чипы связи *Ethernet* и физические носители. Поскольку технология *Ethernet* используется с середины 1970-ых и широко принята во всём мире, то продукты *Ethernet* поддерживает большое количество поставщиков. Используя продукты *Ethernet*, вы не только следуете за общим направлением современной технологии, — у вас есть возможность иметь доступ ко всем устройствам уровня данных из *Internet*. *EtherNet/IP* возник из-за высокого спроса на использование сети *Ethernet* для

приложений управления.

*TCP/IP* — протокол транспортного и сетевого уровней *Internet* и широко связан с сетями *Ethernet* и деловым миром. *TCP/IP* обеспечивает набор сервисов, которые, для совместного использования данных, могут использовать любые два устройства. Поскольку технология *Ethernet* и стандартные блоки протокола, например, *TCP/IP* опубликованы для общественного использования, то стандартизованные сервисные программы и физические носители производятся массово и легко доступны, что дает вам два преимущества:

- известная технология;

- доступность.

Стек протоколов *TCP/IP* был создан на основе *NCP* (*Network Control Protocol*) группой разработчиков под руководством Винтона Серфа в 1972 году. В июле 1976 года Винт Серф и Боб Кан впервые продемонстрировали передачу данных с использованием *TCP* по трём различным сетям. Пакет прошел по следующему маршруту: Сан-Франциско — Лондон — Университет Южной Калифорнии. В конце своего путешествия пакет проделал 150 тысяч км, не потеряв ни одного бита. В 1978 году Серф, Джон Постел и Дэнни Кохэн решили выделить в *TCP* две отдельные функции: *TCP* и *IP* (*Internet Protocol*, межсетевой протокол). *TCP* был ответственен за разбивку сообщения на датаграммы и соединение их в конечном пункте отправки. *IP* отвечал за передачу (с контролем получения) отдельных датаграмм.

*UDP* (*User Datagram Protocol* - протокол пользовательских датаграмм) — также используется совместно с сетью *Ethernet*, один из ключевых элементов *TCP/IP*, набора сетевых протоколов для Интернета. С *UDP* компьютерные приложения могут посылать сообщения (в данном случае называемые датаграммами) другим хостам по *IP*-сети без необходимости предварительного сообщения для установки специальных каналов передачи или путей данных. Протокол был разработан Дэвидом П. Ридом в 1980 году и официально определён в *RFC 768*.

*UDP* использует простую модель передачи, без неявных «рукопожатий» для обеспечения надёжности, упорядочивания или целостности данных. Таким образом, *UDP* предоставляет ненадёжный сервис, и датаграммы могут прийти не по порядку, дублироваться или вовсе исчезнуть без следа. *UDP* подразумевает, что проверка ошибок и исправление либо не нужны, либо должны исполняться в приложении. Чувствительные ко времени приложения часто используют *UDP*, так как предпочтительнее сбросить пакеты, чем ждать задержавшиеся пакеты, что может оказаться невозможным в системах реального времени. При необходимости исправления ошибок на сетевом уровне интерфейса приложение может задействовать *TCP* или *SCTP*, разработанные для этой цели.

*DHCP* (*Dynamic Host Configuration Protocol* — протокол динамической настройки узла) — сетевой протокол, позволяющий компьютерам автоматически получать *IP*-адрес и другие параметры, необходимые для работы в сети *TCP/IP*. Данный протокол работает по модели «клиент-сервер». Для автоматической конфигурации компьютер-клиент на этапе конфигурации сетевого устройства обращается к так называемому серверу *DHCP*, и получает от него нужные параметры. Сетевой администратор может задать диапазон адресов, распределяемых сервером среди компьютеров. Это позволяет избежать ручной настройки компьютеров сети и уменьшает количество ошибок. Протокол *DHCP* используется в

большинстве сетей *TCP/IP*.

## 1.2 Классификация *Ethernet*-пакетов(фреймов)

Фрейм (*Frame* - кадр, битовая цепочка) - пакет канального уровня, минимальная "упаковка" информации, передаваемой по сети.

Фрейм *Ethernet* согласно спецификации *IEEE*802.3 состоит из следующих элементов:

- Преамбула (8 байт): последовательность для синхронизации приемника, заканчивающаяся маркером начала пакета.

- Заголовок (14 байт): содержит *MAC*-адреса источника и приемника (по 6 байт) и двухбайтное поле длины или типа сетевого протокола (в зависимости от типа фрейма).

- Данные (46-1500 байт): содержимое этого поля зависит от типа фрейма.

- Концевик (4 байта): *CRC*-код для контроля достоверности передачи. Фреймы, не удовлетворяющие данной спецификации, считаются ошибочными. По типу ошибки (сочетание реальной длины, контрольной суммы и преамбулы) возможна локализация источника неисправности с точностью до функционального узла канала.

- Семейство фреймов включает в себя *Ethernet_II* и ряд типов, базирующихся на стандарте *IEEE*802.3, имеющих некоторые отличия от классического *Ethernet.*

- Фрейм *Ethernet*_II в третьем элементе заголовка содержит тип сетевого протокола, пославшего этот пакет. В поле данных содержится информация, поступившая от вышестоящего (сетевого) уровня. Этот тип преимущественно используется протоколом *TCP/IP*.

Фреймы, базирующиеся на 802.3, в третьем элементе заголовка содержат длину пакета. К ним относятся: *Ethernet*_802.3, иногда называемый 802.3 *raw* (сырой), поскольку в поле данных информация *LLC*-подуровня не включается. Этот тип применяется по умолчанию в *Novell NetWare* версии 3.11 и ниже.

*Ethernet*_802.2 отличается от 802.3 тем, что в поле данных присутствует заголовок *LLC*-подуровня, содержащий байты *DSAP* и *SSAP* (*Destination* и *Source Service Access Point*), идентифицирующий протоколы сетевого уровня источника и получателя, и 1-2 байта управляющего поля, определяющие требуемый уровень *LLC*-сервиса. Последующие байты содержат информацию, поступившую от вышестоящего (сетевого) уровня. Этот тип применяется по умолчанию в *Novell NetWare* начиная с версии 3.12.

*Ethernet_SNAP* (Sub-*Network Access Protocol*) отличается от 802.2 тем, что к *LLC*-заголовку в поле данных добавлен трехбайтный код организации и двухбайтный код типа сетевого протокола, совпадающий с полем типа заголовка фрейма *Ethernet*_II, например, 0800h - *IP* (*Internet Protocol*), 8137*h* - *Novell NetWare IPX/SPX*. Этот тип тоже применим для протоколов *TCP/IP.*

Тип фрейма указывается при загрузке сетевого драйвера. Приемник может непосредственно получить пакет от источника, если они используют одинаковые типы фреймов. В отличии от других сетевых ОС (*Windows*/*NT*, *Unix*, *LAN Server*), в которых фиксирован только один тип, *Novell* позволяет одновременно использовать несколько типов

фреймов.

# 3 АРХИТЕКТУРА MQTT-СИСТЕМ

## 3.1 Понятие MQTT-брокер

*MQTT* или *Message Queue Telemetry Transport* – это легкий, компактный и открытый протокол обмена данными созданный для передачи данных на удалённых локациях, где требуется небольшой размер кода и есть ограничения по пропускной способности канала. Вышеперечисленные достоинства позволяют применять его в системах *M2M* (Машинно-Машинное взаимодействие) и *IIoT* (Промышленный Интернет вещей).

Также существует версия протокола *MQTT-SN* (*MQTT for Sensor Networks*), ранее известная как *MQTT-S*, которая предназначена для встраиваемых беспроводных устройств без поддержки *TCP/IP* сетей, например, *Zigbee*.

Основные особенности протокола *MQTT*:

- Асинхронный протокол;
- Компактные сообщения;
- Работа в условиях нестабильной связи на линии передачи данных;
- Поддержка нескольких уровней качества обслуживания (*QoS*);
- Легкая интеграция новых устройств;

Протокол *MQTT* работает на прикладном уровне поверх *TCP/IP* (рисунок 3.1) и использует по умолчанию 1883 порт (8883 при подключении через *SSL*).

***Рисунок 3.1 – Протокол *MQTT в файле "Формулы и диаграмы*"***

Обмен сообщениями в протоколе *MQTT* осуществляется между клиентом (*client*), который может быть издателем или подписчиком (*publisher/subscriber*) сообщений, и брокером (*broker*) сообщений (например, *Mosquitto MQTT*).

Издатель отправляет данные на *MQTT* брокер, указывая в сообщении определенную тему, топик (*topic*). Подписчики могут получать разные данные от множества издателей в зависимости от подписки на соответствующие топики.

Устройства *MQTT* используют определенные типы сообщений для взаимодействия с брокером, ниже представлены основные:

- *Connect* – установить соединение с брокером;
- *Disconnect* – разорвать соединение с брокером;
- *Publish* – опубликовать данные в топик на брокере;
- *Subscribe* – подписаться на топик на брокере;
- *Unsubscribe* – отписаться от топика;

### Схема простого взаимодействия между подписчиком, издателем и брокером представлена на рисунке 3.2.

***Рисунок 3.2 - Схема простого взаимодействия между подписчиком, издателем и***

***брокером в файле "Формулы и диаграмы"****

Брокер является основным элементом системы "издатель-подписчик". Он отвечает за прием всех сообщений, их фильтрацию, принятие решения о том, кому интересны эти сообщения, и, в конечном итоге, за пересылку сообщений всем клиентам-подписчикам.

## a. Виды MQTT-брокеров

Среди серверных реализаций брокера можно выделить *IBM WebSphere MQ*; открытое ПО *Mosquitto*; решение, основанное на облачном сервисе *Eurotech Everywhere Device Cloud*; легко масштабируемый и высокопроизводительный открытый сервер *emqttd*, последняя версия (0,17) позволяет обслуживать 1,3 миллиона соединений; брокер *HiveMQ*, обеспечивающий корпоративную безопасность и максимальную масштабируемость.

## b. Краткое описание функционала и способов настройки MQTT-брокеров RSMB и Mosquitto и Amazon

*Mosquitto* - это брокер сообщений с открытым исходным кодом (лицензированный *EPL / EDL*), который реализует протоколы *MQTT* версий 3.1 и 3.1.1. *Mosquitto* легок и подходит для использования на всех устройствах от одноплатных компьютеров с низким энергопотреблением до полных серверов.

Протокол *MQTT* обеспечивает легкий метод осуществления обмена сообщениями с использованием модели публикации / подписки. Это делает его пригодным для обмена сообщениями через Интернет, например, с датчиками малой мощности или мобильными устройствами, такими как телефоны, встроенные компьютеры или микроконтроллеры.

Проект *Mosquitto* также предоставляет библиотеку *C* для реализации клиентов *MQTT* и очень популярных клиентов *MQTT* командной строки *mosquitto_pub* и *mosquitto_sub*.

Начать работу с *Amazon MQ* легко. Запустить брокер сообщений можно за считаные минуты в Консоли управления *AWS* или с помощью интерфейса командной строки (*CLI*).

*Amazon MQ* обеспечивает стартовую настройку и выполнение текущих административных задач: обновление версий ПО и компонентов безопасности, обнаружение сбоев и восстановление. Сервис интегрирован с *Amazon CloudWatch*, что позволяет отслеживать текущие метрики и создавать уведомления о потенциальных проблемах. К примеру, можно вести мониторинг глубины очереди или создавать предупреждения в случае сбоев в доставке сообщений.

## c. Понятие темы (topic)

Топики представляют собой символы с кодировкой *UTF-8*. Иерархическая структура топиков имеет формат «дерева», что упрощает их организацию и доступ к данным. Топики состоят из одного или нескольких уровней, которые разделены между собой символом «/».

Пример топика в который датчик температуры, расположенный в спальной комнате публикует данные брокеру:

/home/living-space/living-room1/temperature

Подписчик может так же получать данные сразу с нескольких топиков, для этого существуют *wildcard*. Они бывают двух типов: одноуровневые и многоуровневые. Для более простого понимания рассмотрим в примерах каждый из них:

- Одноуровневый *wildcard*. Для его использования применяется символ «**+**»

К примеру, нам необходимо получить данные о температуры во всех спальных комнатах:

/home/living-space/**+**/temperature

В результате получаем данные с топиков:

*/home/living-space/living-room1/temperature*

*/home/living-space/living-room2/temperature*

*/home/living-space/living-room3/temperature*

- Многоуровневый *wildcard*. Для его использования применяется символ «**#**»

К примеру, чтобы получить данные с различных датчиков всех спален в доме:

*/home/living-space/***#**

В результате получаем данные с топиков:

*/home/living-space/living-room1/temperature*

*/home/living-space/living-room1/light1*

*/home/living-space/living-room1/light2*

*/home/living-space/living-room1/humidity*

*/home/living-space/living-room2/temperature*

*/home/living-space/living-room2/light1*

a. **Перечень и формат MQTT-команд**

Сообщения *MQTT* содержат обязательный заголовок фиксированной длины (2 байта) и необязательный заголовок переменной длины конкретного сообщения и полезную нагрузку сообщения.

Необязательные поля обычно усложняют обработку протокола. Однако *MQTT* оптимизирован для сетей с ограниченной пропускной способностью и ненадежных сетей (как правило, беспроводных сетей), поэтому дополнительные поля используются для максимально возможного сокращения объема передачи данных.

*MQTT* использует сетевой порядок байтов и битов

***Рисунок 3.3 – Формат сообщения MQTT в файле "Формулы и диаграмы"***

*DUP* - флаг дубликата сообщения. Указывает получателю, что это сообщение, возможно, уже было получено.

*QOS* - указывает уровень гарантии доставки сообщения *PUBLISH*.

0 - Доставка не более одного раза, без гарантий, «Отправь и забудь».

1 - Доставка как минимум один раз, подтвержденная доставка.

2 - Доставка ровно один раз

*Retain* - указывает серверу сохранить последнее полученное сообщение PUBLISH и доставить его в качестве первого сообщения.

**Рисунок 3.4 – Перечень команд *MQTT в файле "Формулы и диаграмы"***

Рассмотрим более детально процесс установления соединения, посылки и приема сообщений (см. рис. 3.5).

**Рисунок 3.5 – Сценарий установления соединения и обмена сообщениями. *в файле "Формулы и диаграмы"***

Установление соединения начинается с передачи от клиента брокеру сообщения *CONNECT*, в котором указываются:

- *ClientId* – уникальный идентификатор для каждого клиента, подключающегося к брокеру;

- *CleanSession* – флаг удаления сохраненных сообщений из предыдущих сессий для данного клиента;

- *Username/Password* – имя пользователя и пароль для идентификации и авторизации клиента.

- *KeepAlive* – временной интервал, регулирующий передачу *ping*-запро-сов и ping-ответов для контроля отключения одной из сторон.

Брокер в ответ посылает клиенту сообщение *CONACK*, состоящее из:

- *Session Present Flag* – указывает существуют ли для данного клиента действующие сессии от предыдущих подключений;

- *Connect Acknowledge Flag* – сообщает клиенту об успешном подключении или о каких-либо ошибках.

После того, как клиент *MQTT* подключен к брокеру, он может публиковать сообщения. Публикация происходит путем отправки брокеру от клиента сообщения *PUBLISH*, где указываются:

- *Topic Name* – название темы, к которой относится данное сообщение. Данное поле является обязательным, так как *MQTT*-брокер принимает решение о пересылке того или иного сообщения клиенту, исходя из тем, на которые клиент подписан;

- Специальные флаги – *QoS, DUP и RETAIN*.

- Полезная нагрузка, где передаются сами данные.

Таким образом, после получения сообщения *PUBLISH* брокер отправляет подтверждение приема публикации (если это задано *QoS*) и пересылает полученное сообщение всем клиентам, которые подписаны на данную тему.

Чтобы получать сообщения с необходимыми данными, *MQTT*-клиент должен сначала подписаться на их получение с помощью сообщения *SUBSCRIBE*. Данное сообщение состоит из

двух частей:

- *Packet Identifier* – необходимо для *QoS* 1 и *QoS* 2;

- *List of Subscriptions* – названия тем, на которые клиент хочет подписаться, и необходимое значение *QoS*.

Стоит отметить, что в протоколе *MQTT* принята иерархическая структура построения тем, поэтому для удобства применяются *wildcard*-символы, благодаря которым подписчик может подписаться на все подтемы данной темы (символ #) либо темы определенного уровня (символ +).

В ответ на сообщение *SUBSCRIBE* брокер отправляет клиенту подтверждение *SUBACK*, в котором сообщает о результате подписки (успешная или нет).

Также клиент может отписаться от темы, которая больше не представляет для него интереса, отправив брокеру сообщение *UNSUBSCRIBE*, в котором будет указана данная тема.

Брокер подтверждает отказ от информации по этой теме сообщением *UNSUBACK*.

# 4 ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

## 4.1 Основные понятия объектно-ориентированного программирования

Раньше программисты, в большинстве случаев, использовали функциональный или процедурный тип программирования. Все программы, большие и маленькие, писались в одном файле. В процессе разработки программы становились сложными и большими, из-за чего у разработчиков возникали проблемы в поддержке и внесении изменений в такие программы. Для решения этой проблемы было придумано объектно-ориентированное программирование.

Объектно-ориентированное программирование (ООП) – методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Существуют чистые и гибридные объектно-ориентированные языки.

Чистые – языки, которые позволяют использовать только одну модель программирования – объектно-ориентированную. Можно объявлять классы и методы, но не можете завести глобальные переменные и обычные функции и процедуры старого типа.

Примерами чистых ОО языков могут быть *Java* и *C#*. На первый взгляд это кажется положительной идеей. Однако она ведет к тому, что вы используете кучу статических методов и статических данных, что не так уж отличается от использования глобальных функций и данных, за исключением более сложного синтаксиса. Чистые ОО языки дают преимущество новичкам в ООП, потому что программист вынужден использовать (и учить) модель ООП. C++ и *Object Pascal*, наоборот, являются типичными примерами гибридных языков, которые позволяют программистам использовать при необходимости традиционный подход C или *Pascal*.

*Smalltalk* расширяет эту идею до уровня «обобъекчивания» таких предопределенных типов данных, как целые и символы, а также языковых конструкций (таких как циклы). Это теоретически интересно, но сильно уменьшает эффективность. *Java* и *C#* останавливаются намного раньше, допуская присутствие простых не ОО типов данных (хотя имеются необязательные классы-обертки и для простых типов).

Ключевыми понятиями ООП являются класс и объект.

Класс – универсальный тип данных, с помощью которого описываются характеристики и возможные действия некоторой сущности.

В классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определенным полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем.

Другим проявлением интерфейсной природы класса является то, что при копировании соответствующей переменной через присваивание, копируется только интерфейс, но не сами данные, то есть класс – ссылочный тип данных. Переменная-объект, относящаяся к заданному классом типу, называется экземпляром этого класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы обеспечить отвечающие природе объекта и решаемой задаче целостность данных объекта, а также удобный и простой интерфейс. В свою очередь, целостность предметной области объектов и их интерфейсов, а также удобство их проектирования, обеспечивается наследованием.

Описав класс, мы можем создать его экземпляр – объект. Объект – это конкретный представитель класса.

ООП основывается на нескольких базовых принципах, а именно:

- Инкапсуляция – свойство системы, позволяющее объединить данные и методы, работающие с ними в классе. В некоторых языках программирования инкапсуляция означает сокрытие внутренней реализации.

- наследование – принцип, позволяющий создавать новый класс на базе другого. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс называется потомком, наследником, дочерним или производным классом.

- полиморфизм подтипов (полиморфизм) – свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. Другой вид полиморфизма – параметрический – в ООП называют обобщенным программированием.

- абстракция данных (абстрагирование) означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных, подразумевая набор значимых характеристик объекта, доступный программе[2].

a. **Технологии программирования, используемые для решения поставленных задач**

Процесс проникновения *IT*-технологий во все сферы общества стал одним из наиболее значимых глобальных процессов современного мира. Численные методы представляют собой классическую область для применения вычислительной техники. До появления первых вычислительных средств теория численных методов обгоняла вычислительные возможности , однако со стремительной эволюцией компьютеров и быстрым развитием технологий программирования ситуация поменялась. Начиная с некоторых пор, уже вычислительные возможности обгоняют теоретические успехи.

Использование современной технологии объектно-ориентированного программирования позволяет рассматривать методы и алгоритмы линейной алгебры, численного анализа, математической физики как самостоятельные объекты. Такой подход дает возможность создать иерархию классов не только алгоритмов методов вычислений, но и иерархию матричных классов и систем управления. Это позволяет модифицировать поведение объектов и придает объектно-ориентированному программированию исключительную гибкость. Происходит создание новых объектов (потомков) на основе уже имеющихся объектов (предков) с передачей их свойств и методов по наследству.

Выделение классов задач и методов их решения, наряду с матричными классами и классами управляющих функционалов, позволяет более четко структурировать программные средства, необходимые для решения задач. Данный прием помогает выразить традиционные математические понятия реальными программными объектами и в конечном итоге достичь желаемой наглядности и выразительности, позволяющей писать сложные прикладные программы в ясной и лаконичной форме, близкой к математической. Применение основных принципов ООП к разработанным алгоритмическим и матричным классификациям может рассматриваться в качестве инструментальной основы для разработки математических библиотек и разнообразных приложений.

С помощью данного подхода созданы следующие библиотеки для решения основных задач вычислительной математики:

- Методы решения нелинейных уравнений и систем.
- Методы решения дифференциальных уравнений и систем.
- Методы интерполяции и аппроксимации.
- Объектно-ориентированное моделирование операторных уравнений.
- Объектно-ориентированная реализация числовых функций и действительных чисел.

Решение математических задач с использованием объектно-ориентированного программирования является современной тенденцией образовательного процесса.

# 5 ПРОЕКТИРОВАНИЕ ГРАФИЧЕСКОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА СРЕДСТВАМИ COM-ОБЪЕКТОВ

*COM* (Component *Object Model*) – технологический стандарт от компании Microsoft, предназначенный для создания программного обеспечения на основе взаимодействующих

компонентов объекта, каждый из которых может использоваться во многих программах одновременно. Стандарт воплощает в себе идеи полиморфизма и инкапсуляции объектно-ориентированного программирования. Он мог бы быть универсальным и платформо-независимым, но закрепился на операционных системах семейства Microsoft Windows. В современных версиях *Windows COM* используется очень широко. На основе *COM* были реализованы технологии: *Microsoft OLE Automation, ActiveX, DCOM, COM+, DirectX u XPCOM.*

Основным понятием, которым оперирует стандарт *COM*, является COM-компонент. Программы, построенные на стандарте *COM*, фактически не являются автономными программами, а представляют собой набор взаимодействующих между собой *COM*-компонентов. Каждый компонент имеет уникальный идентификатор (*GUID*) и может одновременно использоваться многими программами. Компонент взаимодействует с другими программами через *COM*-интерфейсы – наборы абстрактных функций и свойств. Каждый *COM*-компонент должен, как минимум, поддерживать стандартный интерфейс *IUnknown*, который предоставляет базовые средства для работы с компонентом. Интерфейс *IUnknown* включает в себя три метода: *QueryInterface*, *AddRef*, *Release*.

Функции *AddRef* и *Release* отвечают за обычную задачу сопровождения жизненного цикла объекта. При каждом обращении к *AddRef* содержимое счетчика ссылок данного объекта увеличивается на единицу, а при каждом обращении к *Release* – уменьшается. Когда значение счетчика достигает нуля, то объект уничтожается.

*Windows API* предоставляет базовые функции, позволяющие использовать COM-компоненты. Библиотеки *MFC* и, особенно, *ATL*/*WTL* предоставляют более гибкие и удобные средства для работы с *COM*. Библиотека *ATL* от *Microsoft* до сих пор остаётся самым популярным средством создания *COM*-компонентов. Но зачастую *COM*-разработка остаётся ещё довольно сложным делом: программистам приходится вручную выполнять многие рутинные задачи, связанные с *COM* (особенно это заметно в случае разработки на *C++*). Впоследствии (в технологиях *COM+* и особенно *.NET*) *Microsoft* попыталась упростить задачу разработки *COM*-компонентов. Для разработки интерфейса *MQTT*-клиента использовалась встроенная в язык программирования *Java* библиотека *Swing. Swing* - это набор для создания богатого графического интерфейса пользователя (*GUI*) для *Java* программ и апплетов. Он может быть подключен к *JDK 1.1*, как отдельная часть, и вошел в состав инструментария для *Java2*, начиная с *JDK* версии 1.2 и далее. В сравнении с ранее использовавшейся библиотекой *AWT*, библиотека *Swing* имеет ряд преимуществ. Следует выделить основные:

- богатый набор интерфейсных примитивов;
- настраивающийся внешний вид на различных платформах (*look and feel*);
- раздельная архитектура модель-вид (*model-view*);
- встроенная поддержка *HTML.*

Были использованы компоненты :*JTextArea*, *JButton* , *JTextField*, *JLabel*.

***Рисунок 5.1 – Графический интерфейс клиента.***

В нижней части интерфейса располагается область интерфейса с заголовком: «Подключение к серверу». Данная область включает в себя поля для ввода имени пользователя, адреса брокера и порта и кнопки «*Подключение*» и «*Отключение*». На случай

неправильного ввода или неудачного подключения предусмотрено предупреждение об ошибке, в виде всплывающего окна. По умолчанию, в поле адреса брокера и порта добавляется значение «127.0.0.1:1883», а в поле имени пользователя: «*Client1*» Как только пользователь подключается к брокеру кнопка «*Подключение»* становится неактивной, а кнопка *«Отключение»* наоборот – неактивной.

В средней части интерфейса располагается область интерфейса с заголовком: «Подписка на сообщения данной тематики». Данная область включает в себя поля для ввода темы подписки на сообщения, публикации самих сообщений и кнопки *«Подписаться» и «Отписаться»* (На сообщения определённой тематики).

В верхней части интерфейса располагаются поля для ввода темы публикуемого сообщения, ввода самого сообщения и кнопка *«Опубликовать»* (опубликовать сообщение).

В последних двух областях кнопки *«Опубликовать», «Подписаться» и «Отписаться»* являются неактивными до подключения к серверу, что является одним из средств защиты от непредвиденных ситуаций (публикация, подписка и отписка от сообщений, до подключения к серверу).

# 6 АЛГОРИТМ ФУНКЦИОНИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Начинается выполнение программы с ввода данных пользователем. Этими данными являются: имя пользователя (по умолчанию *Client1*) и адрес брокера и порт (по умолчанию 127.0.0.1:1883). После нажатия кнопки «*Подключение»* программа считывает эти данные и пытается подключиться к брокеру. Если при подключении появляется ошибка, то необходимо повторить ввод данных и повторить попытку подключения (рисунок 6.1).

**Рисунок 6.1 – Подключение клиента к брокеру**

Далее пользователю необходимо ввести тему, на которую он хочет подписаться и нажать кнопку *Подписаться*.

Алгоритм данного процесса представлен на рисунке 6.2.

**Рисунок 6.2 – Алгоритм подписки клиента на тему.**

После того, как клиент подключился к брокеру и подписался на необходимую ему тему, он может отправлять сообщения. Чтобы отправить сообщение клиенту, необходимо ввести тему в поле для ввода публикуемой темы и текст сообщения в поле для ввода сообщения и нажать на кнопку *Опубликовать*. Алгоритм данного процесса представлен на рисунке 6.3.

**Рисунок 6.3 – Алгоритм публикации сообщения.**

# 7 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА ОБМЕНА ТЕКСТОВЫМИ СООБЩЕНИЯМИ ПО ПРОТОКОЛУ MQTT

Для создания курсового проекта был использован язык программирования Java. Для создания графического пользовательского интерфейса (*GUI*) была использована библиотека

*Swing.*

Программа состоит из трёх классов: MainClass, AreaOfSubscribe, AreaOfPublish. Каждый класс отвечает за отдельную часть *GUI*. Главный класс также содержит методы для обмена сообщениями по протоколу *MQTT* и связи с брокером.

Примеры методов связи с брокером:

- на примере метода подключения клиента к брокеру:

```
public void Connection(String address, boolean flag)    throws MqttException
  {
    if ((this.newClient != null) &&
      (!address.equals(this.newClient.getConnection())))
    {
      this.newClient.terminate();
      this.newClient = null;
    }
    if (this.newClient == null)
    {
      this.newClient = MqttClient.createMqttClient(address, null);
      this.newClient.registerSimpleHandler(this);
    }
    this.newClient.setRetry(10);
    String str1 = this.id.getText();
    if (isLWTTopicSet())
    {
       this.newClient.connect(str1, this.cleanSessionSelected, (short)this.liveClients, this.topic,
this.QoS, this.data, this.RetainSelected);
    }
    else
    {
      this.newClient.connect(str1, this.cleanSessionSelected, (short)this.liveClients);
    }
  }
```

- на примере метода потери соединения:

```
public void connectionLost()
    throws Exception
  {
    int i = -1;
      JOptionPane.showMessageDialog(null, "Соединение потеряно!... Передподключение",
"WARNING", JOptionPane.PLAIN_MESSAGE);
```

```
        try
        {
          while ((i == -1) && (this.connected))
          {
            try
            {
              synchronized (this.WaitObject)
              {
                this.WaitObject.wait(5000L);
              }
            }
            catch (InterruptedException localInterruptedException) {}
            synchronized (this)
            {
              if (this.connected) {
                try
                {
                  Connection(this.newClient.getConnection(), this.newClient.getPersistence() != null);
                  i = 0;
                }
                catch (MqttException localMqttException)
                {
                  i = -1;
                }
              }
            }
          }
        }
        catch (Exception localException)
        {
              JOptionPane.showMessageDialog(null, "Соединение потеряно!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
          Disconnection();
          throw localException;
        }
        if (this.connected) {
          ConnectButtons(true);
        } else {
          ConnectButtons(false);
        }
      }
```

I. **АНАЛИЗ РЕЗУЛЬТАТОВ РЕШЕНИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ**

Задачей данной курсовой работы – создание графического интерфейса для отправки текстовых сообщений под протоколом *MQTT.*

Проведём анализ выполнения программы.

После включения программы, если клиент не подключён к серверу или введены неправильно данные о клиенте программа выдаст предупреждение об ошибке. Результат на рисунке 8.1-8.2

Рис. 8.1 Сообщение об ошибке при подключении к брокеру в случае его отсутствия

Рис. 8.2 Сообщение об ошибке в случае неправильно введённых данных

Так же на рисунках видно, что нажатие кнопок *Публикация, Подписка, Отписка* и *Отключение* до подключения к серверу неактивны.

Подключаясь к брокеру, пользователь может подписаться на сообщения произвольной темы, а также опубликовать сообщение произвольной темы, что видно на рисунке 8.3

Рис. 8.3 Подписка на определённую тему и публикация сообщения данной темы

После нажатие кнопки *Отключение* происходит отключение от сервера. Также все остальные кнопки переходят в неактивный режим, что является ещё одной защитой для удобной и безопасной работы пользователя. Результат представлен на рисунке 8.4

Рис. 8.4 Отключение от сервера

# 8 ПРИЛОЖЕНИЕ А. Листинг кода программы

```
package newpackage;
```

```
import com.ibm.mqtt.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.EtchedBorder;
public class MainClass  implements ActionListener, MqttSimpleCallback, Runnable
{
  public int liveClients = 30;
  public int QoS = 1;
  public String data = "";
  public String topic = "";
  public boolean cleanSessionSelected = true;
  public boolean RetainSelected = false;
  //
  private JPanel AreaOfConn = new JPanel();
  private JPanel AreaOfPub = new JPanel();
  private JPanel AreaOfSub = new JPanel();
  private JPanel AllCompanents = null;
  private JFrame frame = null;
  private JTextField id;
  private JTextField adress;
  private JButton disconnectButton;
  private JButton connectButton;
  private AreaOfPublish publishArea;
  private AreaOfSubscribe subscribeArea;
  private IMqttClient newClient = null;
  private boolean connected = false;
  private Object WaitObject = new Object();

  protected static final Insets MARGIN_OF_TEXT = new Insets(5, 5, 5, 5);

  public boolean isLWTTopicSet()
  {
    return !this.topic.equals("");
  }

  public static void main(String[] args)
  {
    JFrame mainJFrame = null;
    MainClass MQTTClientFrame = new MainClass();
    mainJFrame = MQTTClientFrame.getJFrame();    mainJFrame.setSize(550, 650);
```

```
  mainJFrame.setLocation(440, 60);
  mainJFrame.setResizable(true);
  MQTTClientFrame.CreateFrame(mainJFrame.getContentPane());
  mainJFrame.setVisible(true);
  mainJFrame.addWindowListener(new WindowAdapter()
  {
    @Override
    public void windowClosing(WindowEvent paramAnonymousWindowEvent)
    {
      System.exit(0);
    }
  });
}
private JFrame getJFrame()
{
  if (this.frame == null) {
    this.frame = new JFrame();
  }
  this.frame.setTitle("MQTT Клиент");
  return this.frame;
}
protected void CreateFrame(Container paramContainer)
{
     try
  {      UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
  }
  catch (Exception localException1)
  {
    localException1.printStackTrace();
  }
  this.AreaOfConn.setLayout(new GridLayout(3, 1));
  this.AreaOfConn.setBorder(new EtchedBorder());
  this.AllCompanents = new JPanel(new BorderLayout());
  this.AllCompanents.add(this.AreaOfConn, "South");
  //
  JPanel userPanel = new JPanel();
  userPanel.setLayout(new GridLayout(2, 1));
  userPanel.add(this.AreaOfPub);
  userPanel.add(this.AreaOfSub);
  this.AllCompanents.add(userPanel);
  ///*****///
```

```
JPanel ConnectionPan1 = new JPanel();
JPanel ConnectionPan2 = new JPanel();
JPanel ConnectionPan3 = new JPanel();
//

JLabel connection_label = new JLabel("Подключение к серверу");
Font Font = connection_label.getFont();
connection_label.setFont(new Font(Font.getName(), 1, Font.getSize() + 1));
//
this.adress = new JTextField("127.0.0.1:1883");
this.adress.setBackground(Color.WHITE);
this.adress.setPreferredSize(new Dimension(125, 35));
this.adress.setEditable(true);
//
this.id = new JTextField("Clien1");
this.id.setBackground(Color.WHITE);
this.id.setPreferredSize(new Dimension(100, 35));
this.id.setEditable(true);
//
this.connectButton = new JButton("Подключение");
this.disconnectButton = new JButton("Отключение");
//
this.disconnectButton.setEnabled(false);
this.connectButton.addActionListener(this);
this.disconnectButton.addActionListener(this);
//
JLabel id_label = new JLabel("ID: ");
JLabel adress_label = new JLabel("Adress: ");
//
ConnectionPan1.add(connection_label);
ConnectionPan2.add(id_label);
ConnectionPan2.add(this.id);
ConnectionPan2.add(adress_label);
ConnectionPan2.add(this.adress);
ConnectionPan3.add(this.disconnectButton);
ConnectionPan3.add(this.connectButton);
//
this.AreaOfConn.add(ConnectionPan1);
this.AreaOfConn.add(ConnectionPan2);
this.AreaOfConn.add(ConnectionPan3);
//
```

```
      this.publishArea = new AreaOfPublish(this.AreaOfPub, this);
      this.subscribeArea = new AreaOfSubscribe(this.AreaOfSub, this);
      paramContainer.add(AllCompanents);
    }
    @Override
    public void run()
    {
      String str1 = this.adress.getText();
      try
      {
        if (!str1.contains("://")) {
          str1 = "tcp://" + str1;
        }
        Connection(str1, false);

        this.connected = true;
        ConnectButtons(true);
      }
      catch (NumberFormatException localNumberFormatException)
      {
            JOptionPane.showMessageDialog(null, "Ошибка подключения!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
      }
      catch (Exception localException)
      {
            JOptionPane.showMessageDialog(null, "Ошибка подключения!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
      }
      if (!this.connected)
      {
        ConnectButtons(false);
      }
    }
    public void Connection(String address, boolean flag)    throws MqttException
    {
      if ((this.newClient != null) &&
        (!address.equals(this.newClient.getConnection())))
      {
        this.newClient.terminate();
        this.newClient = null;
      }
```

```
        if (this.newClient == null)
        {
          this.newClient = MqttClient.createMqttClient(address, null);
          this.newClient.registerSimpleHandler(this);
        }
        this.newClient.setRetry(10);
        String str1 = this.id.getText();
        if (isLWTTopicSet())
        {       this.newClient.connect(str1, this.cleanSessionSelected, (short)this.liveClients, this.topic,
this.QoS, this.data, this.RetainSelected);
        }
        else
        {
          this.newClient.connect(str1, this.cleanSessionSelected, (short)this.liveClients);}

      }
      public void Disconnection()
      {
        this.connected = false;
        synchronized (this.WaitObject)
        {
          this.WaitObject.notify();
        }
        if (this.newClient != null) {
          try
          {
            this.newClient.disconnect();
          }
          catch (Exception ex)
          {
                JOptionPane.showMessageDialog(null, "Ошибка при отключении!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
            System.exit(1);
          }
        }
        ConnectButtons(false);
      }

      public void publish(String paramString, byte[] message, int qos, boolean isRetained)
        throws Exception
      {
```

```
        try
        {
          this.newClient.publish(paramString, message, qos, isRetained);
        }
        catch (MqttException ex)
        {
              JOptionPane.showMessageDialog(null, "Ошибка публикации!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
          throw ex;
        }
      }
      public void subscription(String topic, int qos, boolean sub)
      {
        try
        {
          String[] arrOfString = new String[1];
          int[] arrOfInt = new int[1];
          arrOfString[0] = topic;
          arrOfInt[0] = qos;
          if (sub) {
            this.newClient.subscribe(arrOfString, arrOfInt);

          } else {
            this.newClient.unsubscribe(arrOfString);
          }
        }
        catch (Exception localException)
        {
              JOptionPane.showMessageDialog(null, "Ошибка подписки!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
        }
      }
      @Override
      public void actionPerformed(ActionEvent paramActionEvent)
      {
        switch (paramActionEvent.getActionCommand())
        {
          case "Подключение":
            if (!this.connected)
            {this.connectButton.setEnabled(false);
```

```
                this.connected = false;
                new Thread(this).start();
              }       break;
            case "Отключение":
              Disconnection();
    //           //
        }
      }
      @Override
      public void connectionLost()
        throws Exception
      {
        int i = -1;
          JOptionPane.showMessageDialog(null, "Соединение потеряно!... Передподключение",
"WARNING", JOptionPane.PLAIN_MESSAGE);
        try
        {
          while ((i == -1) && (this.connected))
          {
            try
            {
              synchronized (this.WaitObject)
              {
                this.WaitObject.wait(5000L);
              }
            }
            catch (InterruptedException localInterruptedException) {}
            synchronized (this)
            {
              if (this.connected) {
                try
                {
                  Connection(this.newClient.getConnection(), this.newClient.getPersistence() != null);
                  i = 0;
                }
                catch (MqttException localMqttException)
                {
                  i = -1;
                }
              }
            }
```

```
        }
       }
       catch (Exception localException)
       {
            JOptionPane.showMessageDialog(null, "Соединение потеряно!", "ERROR",
JOptionPane.PLAIN_MESSAGE);
        Disconnection();
        throw localException;
       }
       if (this.connected) {
        ConnectButtons(true);
       } else {
        ConnectButtons(false);
       }
      }

      @Override
      public void publishArrived(String paramString, byte[] paramArrayOfByte, int paramInt, boolean
paramBoolean)
      {
          this.subscribeArea.updateReceivedData(paramString, paramArrayOfByte, paramInt,
paramBoolean);
      }

      public void updateSubscribeTopicList(String paramString)
      {
       this.subscribeArea.updateTopicList(paramString);
      }
      public void updatePublishTopicList(String paramString)
      {
       this.publishArea.updateTopicList(paramString);
      }

      public boolean updateComboBoxList(JComboBox paramJComboBox, String paramString)
      {
       int i = paramJComboBox.getItemCount();
       int j = 0;
       if (paramString == null) {
        return false;
       }
       for (int k = 0; k < i; k++) {
```

```
      if (((String)paramJComboBox.getItemAt(k)).equals(paramString))
      {
       j = 1;
       break;
      }
    }
    if (j == 0)
    {
     paramJComboBox.addItem(paramString);
     return true;
    }
    return false;
  }

  private void ConnectButtons(boolean paramBoolean)
  {
    this.publishArea.enableButtons(paramBoolean);
    this.subscribeArea.enableButtons(paramBoolean);
    this.disconnectButton.setEnabled(paramBoolean);
    this.connectButton.setEnabled(!paramBoolean);
  }
}
package newpackage;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.EtchedBorder;
public class AreaOfPublish implements ActionListener, DocumentListener
{
  private JPanel PublishPanel;
  private JButton publishButton;
  private JTextArea data;
  private JTextField topic;
  private MainClass MQTTComp = null;

  public AreaOfPublish(JPanel PublishPanel, MainClass mqttComp)
  {
    this.PublishPanel = PublishPanel;
    this.PublishPanel.setBorder(new EtchedBorder());
    this.MQTTComp = mqttComp;
```

```
  create();
}
public void create()
{
  this.PublishPanel.setLayout(new BorderLayout());
   //
  JPanel PublishPanel1 = new JPanel();
  JPanel PublishPanel2 = new JPanel();
  JPanel PublishPanel3 = new JPanel();
  JPanel PublishPanel4 = new JPanel();
  JPanel panel = new JPanel();
  //
  JLabel PubishLabel = new JLabel("Публикация сообщений данной тематики");
  PubishLabel.setFont(new Font("Times new roman", Font.BOLD, 16));
  //
  PublishPanel1.setLayout(new BoxLayout(PublishPanel1, 0));
  PublishPanel1.add(new JLabel(" Тема публикации:"));
  this.topic = new JTextField();
  this.topic.setEditable(true);
  PublishPanel1.add(this.topic);
  panel.setLayout(new BoxLayout(panel, 0));
  //
  this.data = new JTextArea(3, 30);
  this.data.setMargin(MainClass.MARGIN_OF_TEXT);
  this.data.setFont(new Font("Times new roman", Font.PLAIN, 15));
  this.data.setBackground(Color.WHITE);

   //
  this.publishButton = new JButton("Опубликовать");
  this.publishButton.setEnabled(false);
  this.publishButton.addActionListener(this);
  PublishPanel3.add(this.publishButton);
  PublishPanel4.add(PublishPanel3);
  //
  PublishPanel2.setLayout(new GridLayout(3, 1));
  PublishPanel2.add(PubishLabel);
  PublishPanel2.add(PublishPanel1);
  PublishPanel2.add(panel);
  //
  this.PublishPanel.add(PublishPanel2, "North");
  this.PublishPanel.add(new JScrollPane(this.data), "Center");
```

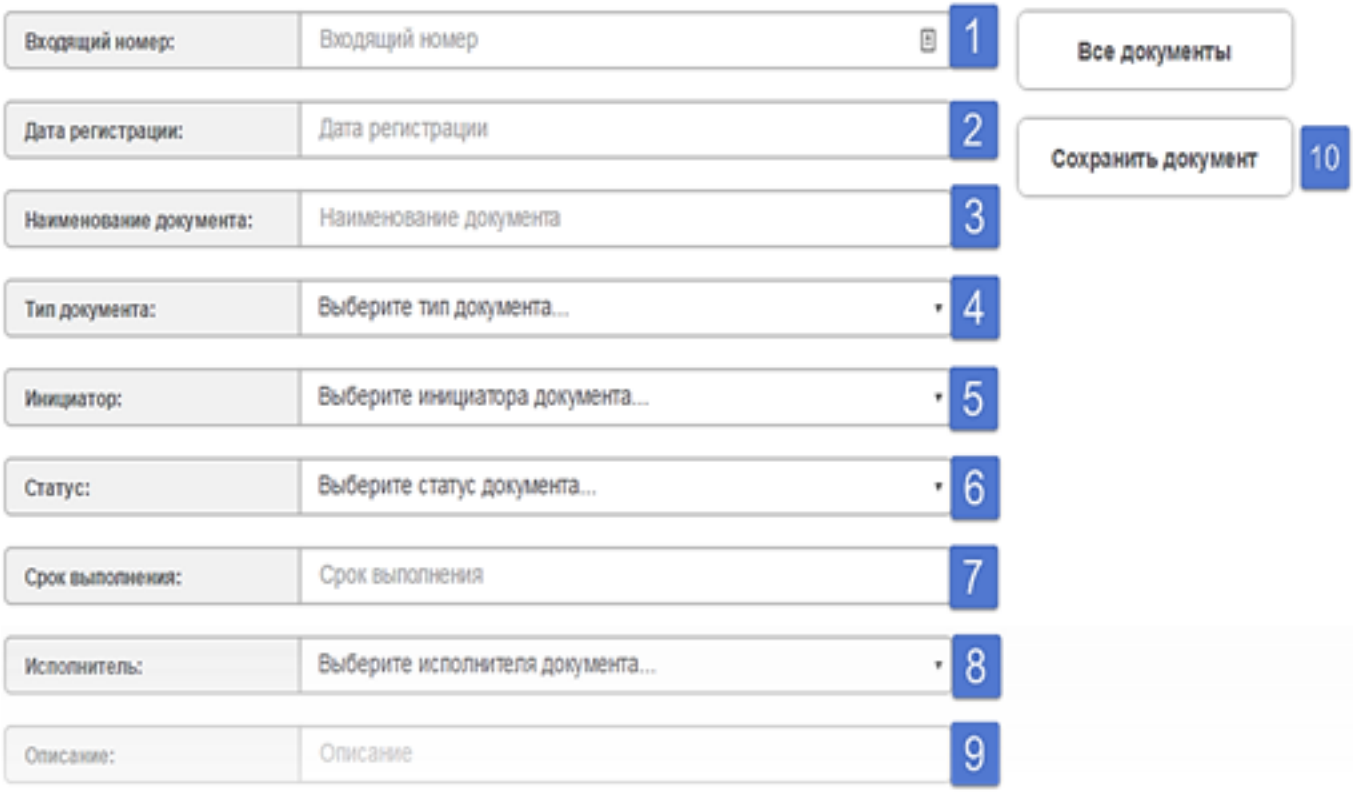
```
  this.PublishPanel.add(PublishPanel4, "East");
  //
}

public boolean updateTopicList(String paramString)
{
  return true;
}

@Override
public void actionPerformed(ActionEvent paramActionEvent)
{
  if (paramActionEvent.getActionCommand().equals("Опубликовать"))
  {
    String Object1 = this.data.getText();
    String Object2 = this.topic.getText();
    if (updateTopicList(Object2)) {
      this.MQTTComp.updateSubscribeTopicList(Object2);
    }
    try
    {
      this.MQTTComp.publish(Object2, Object1.getBytes(), 1, false);
    }
    catch (Exception localException)
    {
    }
  }
}

public void enableButtons(boolean b)
{
  this.publishButton.setEnabled(b);
}

@Override
public void changedUpdate(DocumentEvent paramDocumentEvent) {}

@Override
public void insertUpdate(DocumentEvent paramDocumentEvent) {}

@Override
```

```
  public void removeUpdate(DocumentEvent paramDocumentEvent) {}
}
package newpackage;
import javax.swing.*;
import javax.swing.border.EtchedBorder;
import java.awt.*;
import java.awt.event.*;

public class AreaOfSubscribe  implements ActionListener
{
  private MainClass mqttMgr = null;
  private JPanel subscribePanel;
  private JTextField topic;
  private JTextArea receivedData;
  private JLabel subcribeLabel = null;
  private JButton subscribeButton;
  private JButton unsubscribeButton;
  public AreaOfSubscribe(JPanel panel, MainClass Frame)
  {
    this.subscribePanel = panel;
    this.subscribePanel.setBorder(new EtchedBorder());
    this.mqttMgr = Frame;
    this.subscribePanel.setLayout(new BorderLayout());
    this.topic = new JTextField();
    this.topic.setEditable(true);
    this.receivedData = new JTextArea(3, 30);
    this.receivedData.setMargin(MainClass.MARGIN_OF_TEXT);
    this.receivedData.setFont(new Font("Times new roman", Font.PLAIN,15));
    this.receivedData.setEditable(false);
    this.receivedData.setBackground(Color.WHITE);
    JPanel panel1 = new JPanel();
    panel1.setLayout(new BoxLayout(panel1, 0));
    panel1.add(new JLabel(" Тема подписки:"));
    panel1.add(this.topic);
    JPanel panel2 = new JPanel();
    panel2.setLayout(new BoxLayout(panel2, 0));

    JPanel panel3 = new JPanel();
    panel3.setLayout(new GridLayout(3, 1));
    this.subcribeLabel = new JLabel("Подписка на сообщения данной тематики");
    this.subcribeLabel.setFont(new Font("Serif", Font.BOLD,12));
```

```
  Font localFont = this.subcribeLabel.getFont();
  this.subcribeLabel.setFont(new Font(localFont.getName(), 1, localFont.getSize() + 3));
  panel3.add(this.subcribeLabel);
  panel3.add(panel1);
  panel3.add(panel2);
  JPanel localJPanel4 = new JPanel();
  JPanel JPanel5 = new JPanel();
  JPanel5.setLayout(new GridLayout(2, 1));
  this.subscribeButton = new JButton("Подписаться");
  this.subscribeButton.setEnabled(false);
  this.subscribeButton.addActionListener(this);
  this.unsubscribeButton = new JButton("Отписаться");
  this.unsubscribeButton.setEnabled(false);
  this.unsubscribeButton.addActionListener(this);
  JPanel5.add(this.subscribeButton);
  JPanel5.add(this.unsubscribeButton);

  localJPanel4.add(JPanel5);
  this.subscribePanel.add(panel3, "North");
  this.subscribePanel.add(new JScrollPane(this.receivedData), "Center");
  this.subscribePanel.add(localJPanel4, "East");
}

public boolean updateTopicList(String topic)
{
  return true;
}

@Override
public void actionPerformed(ActionEvent e)
{
  String strTopic = this.topic.getText();
  if (updateTopicList(strTopic)) {
    this.mqttMgr.updatePublishTopicList(strTopic);
  }
    switch (e.getActionCommand()) {
        case "Подписаться":
            this.mqttMgr.subscription(strTopic, 1, true);
            break;
        case "Отписаться":
            this.mqttMgr.subscription(strTopic, 0, false);
```

```
            break;
        }
    }
    public void updateReceivedData(String topic, byte[] data, int QoS, boolean retained)
    {
      this.receivedData.setText(new String(data));
    }
    public void enableButtons(boolean b)
    {
      this.subscribeButton.setEnabled(b);
      this.unsubscribeButton.setEnabled(b);
    }
}
```

# 9 ПРОДОЛЖЕНИЕ Б. Скриншоты сайта системы Антиплагиат




# 10 8. АНАЛИЗ РЕЗУЛЬТАТОВ РЕШЕНИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ

Задачей данной курсовой работы – создание графического интерфейса для отправки

текстовых сообщений под протоколом *MQTT.*

Проведём анализ выполнения программы.

После включения программы, если клиент не подключён к серверу или введены неправильно данные о клиенте программа выдаст предупреждение об ошибке. Результат на рисунке 8.1-8.2

***Рис. 8.1 Сообщение об ошибке при подключении к брокеру в случае его отсутстви***

***Рис. 8.2 Сообщение об ошибке в случае неправильно введённых данных***

Так же на рисунках видно, что нажатие кнопок *Публикация, Подписка, Отписка* и *Отключение* до подключения к серверу неактивны.

Подключаясь к брокеру, пользователь может подписаться на сообщения произвольной темы, а также опубликовать сообщение произвольной темы, что видно на рисунке 8.3

***Рис. 8.3 Подписка на определённую тему и публикация сообщения данной темы***

После нажатие кнопки *Отключение* происходит отключение от сервера. Также все остальные кнопки переходят в неактивный режим, что является ещё одной защитой для удобной и безопасной работы пользователя. Результат представлен на рисунке 8.4

***Рис. 8.4 Отключение от сервера***

# Заключение

В данной курсовой работе была разработана программа MQTT-клиента на языке программирования Java. Для создания графического пользовательского интерфейса была использована библиотека Swing. Программа позволяет пользователю подключаться к брокеру, подписываться на рассылку по интересующей пользователя теме и отправлять сообщения определённой темы. В программе имеется защита на работу с публикаций сообщений, подпиской и отпиской от тематики, в случае отсутствия соединения, множество проверок на корректность ввода данных и постоянная попытка подключения к брокеру в случае потери соединения.

# Список использованных источников

1. [url] **Протокол MQTT. Особенности, варианты применения, основные процедуры MQTT Protocol. [Электронный ресурс]. – Режим доступа:** http://lib.tssonline.ru/articles2/fix-corp/protokol-mqtt-osobennosti-varianty-primeneniya-osnovnye-protsedury-mqtt-protocol
2. [url] **Что такое MQTT и для чего он нужен в IIoT? Описание протокола MQTT [Электронный ресурс]. – Режим доступа:** https://ipc2u.ru/articles/prostye-resheniya/chto-takoe-mqtt/
3. [url] **Задачи теории массового обслуживания. Классификация систем массового обслуживания [Электронный ресурс]. – Режим доступа:** http://ru.itmodeling.wikia.com
4. [url] **Задачи теории массового обслуживания [Электронный ресурс]. – Режим доступа:** https://studfiles.net/preview/2419983/page:2/

5. [url] **Причины популярности Интернета вещей [Электронный ресурс]** https://habr.com/post/388231/
6. [url] **Одноканальная СМО с ограниченной очередью [Электронный ресурс].** http://e-biblio.ru/book/bib/06_management/teor_mass_obslug/158.9.16.html
7. [url] **Семейство протоколов TCP/IP,IP,UDP, ICMP, DHCP, ARP [Электронный ресурс].** http://main.tpkelbook.com/
8. [url] **Что такое фреймы Ethernet и чем они различаются? [Электронный ресурс].** http://www.i-assembler.ru/1a/glava-04-chto-takoe-frejmy-ethernet-i-chem-oni.htm

# Приложения

1. [электронный документ] 5ebd1b328ef78_Пояснительная записка Зеньковский.docx
2. [электронный документ] 5ebd53da32300_Формулы и рисунки.docx