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

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

РАБОТА ПРОВЕРЕНА
Рецензент
_____
«___» _____ 2019 г.

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_____ Г.И. Радченко
«___» _____ 2019 г.

Модернизация систем охранной и пожарной сигнализаций умного дома

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

Руководитель работы,
к.т.н., доцент каф. ЭВМ
_____ И.Л. Кафтанников
«___» _____2019 г.

Автор работы,
студент группы КЭ-222
_____ Д.И. Сергеев
«___» _____2019 г.

Нормоконтролёр,
ст. преп. каф. ЭВМ
_____ С.В. Сяськов
«___» _____2019 г.

Челябинск-2019

Аннотация

Д.И. Сергеев. Модернизация систем охранной и пожарной сигнализаций умного дома. – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШЭКН; 2019, 80 с., 13 ил., библиогр. список – 36 наим.

В рамках выпускной квалификационной работы производится детальный анализ современных подходов к построению систем умного дома, в частности интеграции с системами охранной и пожарной сигнализации. Формируется список требований к современным системам умного дома. Разрабатывается архитектура системы, отвечающая всем представленным требованиям и при этом, позволяющая интегрировать и использовать элементы инфраструктуры уже существующих подсистем, с обоснованием выбранных решений. Приводится реализация основных компонентов системы для доказательства работоспособности и тестирования предложенного решения. Формируется план по возможному дальнейшему развитию системы.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время широкое распространение получило понятие Internet of things (IoT) или интернета вещей, которое в широком смысле охватывает едва ли не все, что каким-либо образом связано с Интернетом. В более узком смысле понятие может обозначать всё множество устройств, которые используют сеть Интернет для передачи информации и взаимодействия друг с другом. При этом к IoT относят любые типы устройств, от простейших сенсоров и датчиков до смартфонов и носимых устройств. Объединение таких устройств в автоматизированные системы позволяет не только собирать огромное количество информации, но и производить их анализ и обработку с целью выработки конкретных действий, направленных на решение поставленной задачи.

Со стороны бизнеса распространение IoT предлагает возможности повышения производительности, снижения издержек, экономию времени и денежных средств, во много позволяя по-новому взглянуть на предоставляемые сервисы и производимые продукты, точнее определить потребности потребителей и выявить новые возможности дальнейшего развития.

Широкое распространение IoT получил и в социальной сфере. Сюда можно отнести использование IoT в медицине, логистике, энергетике, метеорологии, в задачах экологического мониторинга, наблюдения за окружающей средой, в сфере интеллектуализации человеческого окружения. При рассмотрении последнего чаще всего выделяют понятия умного города и умного дома.

Концепция умного города предполагает наличие взаимосвязанной системы коммуникативных и информационных технологий с интернетом вещей, благодаря которой упрощается управление внутренними процессами

города и улучшается уровень жизни населения. Развитие города в русле предлагаемой концепции очень непростая и многофакторная задача, так как включает в себя развитие сфер управления, экономики и финансов, развитие инфраструктуры, а также требует активного участия жителей в жизни города.

С данной точки зрения концепция умного дома более локальна и проста в реализации в отдельно взятом случае. Она предполагает наличие высокотехнологичной системы, которая позволяет объединить все используемые коммуникации, поставить их под централизованное управление и предоставить пользователю удобные средства для настройки и управления. Наиболее перспективные направления развития в области умного дома приведены в таблице 1.

Таблица 1 – Перспективы развития категорий «умного дома»

«Умный дом» по категориям, 2019 – 2023 (кол-во устройств – млн. штук)					
Категория	**2019 устройства**	**2019 объём рынка**	**2023 устройства**	**2023 объём рынка**	**2019 – 2023 CAGR***
Мониторинг и безопасность	140,3	16,8%	351,7	22,6%	25,8%
Освещение	56,9	6,8%	183,2	11,8%	34,0%
Прочее	114,3	13,7%	269,4	17,3%	23,9%
Голосовое управление	144,3	17,3%	240,1	15,4%	13,6%
Управление температурой	18,8	2,3%	37,5	2,4%	18,8%
Видеонаблю-дение	358,1	43,0%	475,4	30,5%	7,3%
Всего	832,7	100,0%	1557,4	100,0%	16,9%
Источник: IDC Worldwide Quarterly Smart Home Device Tracker, 29 марта 2019 года ***CAGR – Compound Annual Growth Rate – Совокупный среднегодовой темп роста**					

Согласно приведённым данным одним из наиболее активно развивающихся направлений в системах умного дома является мониторинг и безопасность.

Целью представленной выпускной квалификационной работы является разработка архитектуры и реализация расширяемой модульной системы умного дома с открытым исходным кодом, возможностью подключения большого количества типов оконечных устройств и интеграции существующих систем охранной и пожарной сигнализации. При этом, система должна удовлетворять высоким требованиям по надёжность, безопасности и обращению информации, предъявляемым к интегрируемым подсистемам.

Для достижения поставленной цели, необходимо решить следующие поставленные задачи:

1. Провести обзор вариантов существующих систем пожарной и охранной сигнализации.
2. Провести обзор аналогов – систем умного дома, предоставляющих средства мониторинга и безопасности.
3. Выделить недостатки рассмотренных систем и сформировать список требования к новой системе.
4. Разработать расширяемую модульную архитектуру системы, удовлетворяющую полученным требованиям.
5. Выбрать протоколы, аппаратные и программные средства для реализации системы удовлетворяющей поставленным требованиям.
6. Реализовать основные компоненты системы для тестирования и демонстрации работы полученного решения.
7. Сформировать план возможного дальнейшего развития полученного решения.

1. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Реализация системы умного дома – непростая задача, так как требует интеграции большого количества разнообразных систем и устройств. Чаще всего к управляемым элементам умного дома относят:

- системы отопления дома;
- системы вентиляции и кондиционирования;
- системы охранной и пожарной сигнализации;
- системы контроля доступа;
- системы видеонаблюдения;
- контроль аварийных ситуаций (утечки воды или газа, аварии электросетей, отключение Интернета и т.д.);
- управление внутренним и наружным освещением;
- управление внутренними и наружными силовыми механизмами (воротами, шлагбаумами, приводами штор, жалюзи и т.д.);
- контроль над энергопотреблением, ограничение пиковых нагрузок и распределение нагрузок по фазам питающей сети;
- управление источниками резервного и аварийного питания;
- управление системами полива.

Приведённый список не является исчерпывающим и может быть дополнен под нужды человека в каждом отдельном случае. Помимо управления система умного дома должна выполнять мониторинг состояния и контроль работоспособности всех подчиненных систем, иметь возможность удаленной настройки и управления всеми узлами системы [1]. Также с развитием системы умного дома и увеличением количества интегрированных подсистем остро встают вопросы надежности систем, их производительности и безопасности [2]. Надежность определяет возможность системы бесперебойно выполнять свои

функции в течении продолжительного периода времени, а также иметь средства восстановления системы в случае сбоев. Производительность характеризуется способностью систем управления поддерживать работоспособность системы для всех возможных сценариев использования с обеспечением требуемого уровня надёжности системы. Под безопасностью обычно понимается способность системы противостоять несанкционированному доступу и невозможность третьими лицами получения информации о работе системы, её структуре или любой другой приватной информации. Три указанные характеристики системы в совокупности с простыми, функциональными и понятными для пользователя интерфейсами определяют качество системы умного дома.

В настоящее время существует множество систем, реализующих концепцию умного дома от различных известных разработчиков программного и аппаратного обеспечения. Далее будут рассмотрены некоторые из них, с целью выявления достоинств и недостатков. С учетом того, что для системы охранной сигнализации наиболее приоритетными является вопросы надежности и безопасности, при рассмотрении существующих решений упор будет сделан на освещение этих вопросов.

Помимо реализаций охранных и пожарных систем, существующих в рамках концепции умного дома, будет рассмотрен пример традиционного подхода к организации системы в виде полностью самостоятельного решения, которое наиболее распространено на сегодняшний день, с целью изучения возможностей построения новой платформы с использованием уже существующих оконечных датчиков и сенсоров.

1.1. СИСТЕМА «ОРИОН» КОМПАНИИ БОЛИД

Традиционный подход построения охранных и пожарных систем предполагает наличие главного устройства управления с возможностью подключения к нему сигнализационных шлейфов. Шлейф включает в себя от одного до нескольких устройств соединенных последовательно. Большинство датчиков имею 4 вывода: 2 для питания устройства и 2 реализующих «сухой контакт», то есть выводы, которые замыкаются в рабочем режиме датчика и размыкаются в случае его срабатывания, например, обнаружения движения, соответствующим сенсором. При параллельном соединение датчиков срабатывание любого из них может быть распознано управляющим устройством как сигнал тревоги и каким-либо образом обработано. Обычно управляющее устройство имеет программируемые выводы реле для подключения выносных устройств: управление аварийным освещением, сиренами и звуковой индикацией, лампами состояния системы и т.д.

Примером такой системы может служить охранная сигнализация, построенная на компонентах фирмы Болид из серии «Орион» [3]. В качестве устройства управления в данном случае используется приемно-контрольные охранно-пожарные пульты «Сигнал-10М» или «Сигнал-20М», которые имеет соответственно по 10 или 20 выводов для подключения сигнализационных шлейфов. Для подключения большего количества датчиков предполагается использовать сетевой контроллер, например, пульт контроля и управления охранно-пожарный С2000М, в которому могут подключатся несколько приборов «Сигнал-20М». Для объединения приборов используется проводная линия связи RS-485 и протокол канального уровня Modbus RTU. Для передачи сообщений срабатывании сигнализации используются выносные приборы передачи извещений, например, телефонный информатор С2000-ИТ. Данные

приборы подключаются в систему с помощью той же шины Modbus, а в качестве канала передачи сигналов используют телефонную линию. Общая схема организации сети в таком случае может иметь вид, представленный на рисунке 1.

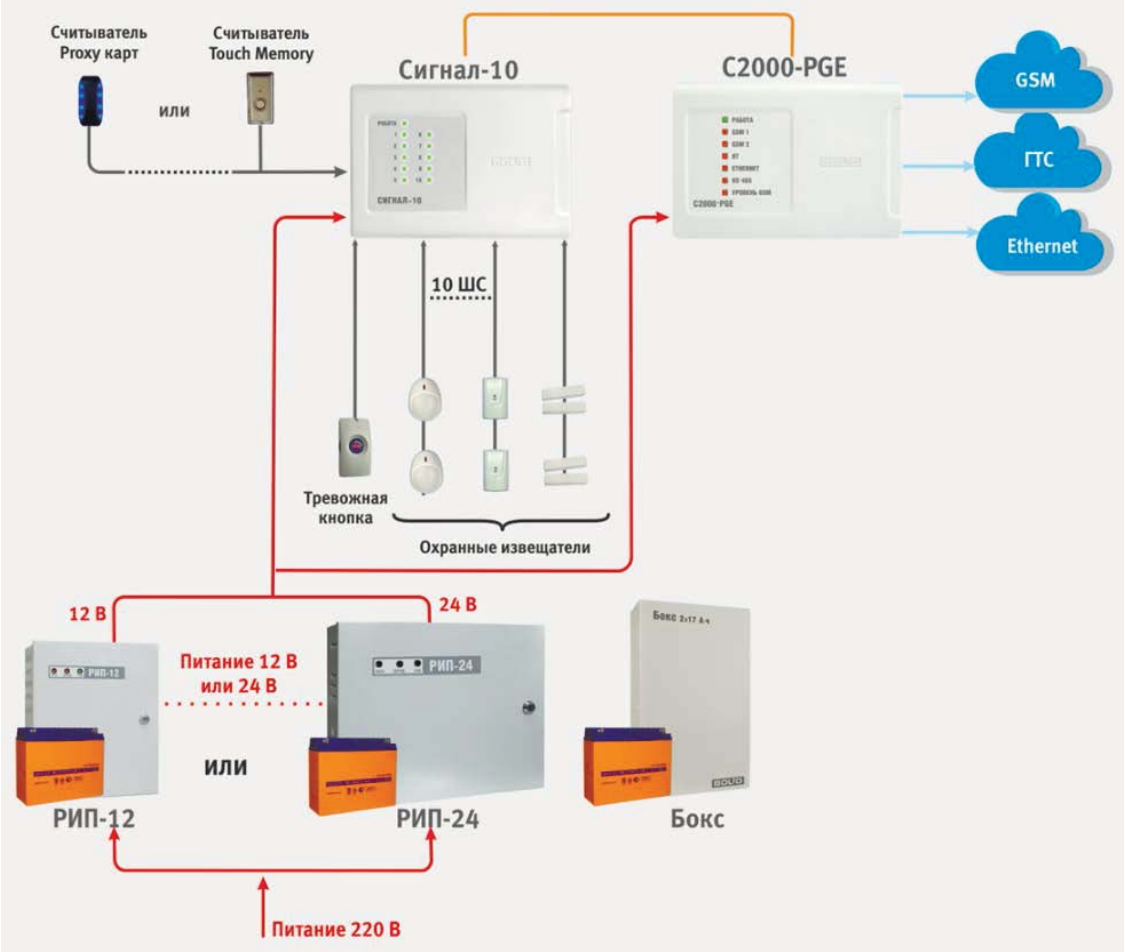

Рисунок 1 – Сеть охранно-пожарной сигнализации «Орион», производства фирмы Болид

Очевидными достоинствами такой системы являются:

– надёжность системы (при грамотном проектировании и монтаже системы);

– безопасность (в силу отсутствия возможности подключения к системе извне);
– развитая инфраструктура и наличие большого количества совместимых устройств;
– универсальность относительно используемых оконечных устройств;
– контроль работоспособности оконечных устройств;
– простота организации бесперебойного питания системы в том числе питания оконечных устройств.

На ряду с указанными достоинствами существует ряд существенных недостатков:

– возможности системы по переконфигурированию или подключению новых устройств ограничены изначально заложенными запасами аппаратных средств и проводными интерфейсами;
– расширение системы происходит экстенсивно и требует подключения новых управляющих устройств;
– отсутствует подключение устройств «на лету», требуется настройка с помощью компьютера и специального ПО;
– настройка и монтаж системы требуют специальных навыков, которыми пользователь системы, в общем случае, не обладает;
– отсутствует возможность удаленного управления и настройки системы;
– могут возникнуть сложности по поиску неисправности между оконечными устройствами и устройством управления, а частности при повреждении кабельных линий связи;
– отсутствует адресность (регистрируется неисправность всего шлейфа, который может содержать большое количества устройств);

– может возникнуть ситуация, когда отсутствует доступ к кабелям, что ведёт к невозможности восстановления работы системы без проведения строительных работ;
– нет стандартных решений по интеграции системы охранной и пожарной сигнализации в систему более высокого уровня, например, для добавления управления ей в пользовательские сценарии управления умным домом.

Кроме указанных недостатков, необходимость прокладки кабельных линий и наличия специальных знаний для проектирования, установки и настройки систем обуславливают их высокую стоимость.

1.2. СИСТЕМА УМНОГО ДОМА XIAOMI

Одним из мировых лидеров производства электронных устройств является китайская компания Xiaomi. Компания имеет несколько дочерних компаний, которые специализируются на системах умного дома, в частности, выпускает систему охранной сигнализации Smart Home Set под брендами MiJia и Aquara. Набор состоит из следующих компонентов:

– концентратор Mi Control Hub;
– выключатель Mi Wireless Switch;
– датчики открытия окон и дверей Mi Window & Door Center;
– датчики движения Mi Motion Sensor.

Схема организации сети для такой системы приведена на рисунке 2. Главным устройством в системе является Mi Control Hub, которое функционально объединяет в себе динамик, датчик освещённости и RGB диодную подсветку для индикации и уведомлений. Конструктивно устройство

имеет вилку питания и предназначено для подключения к стандартной сети переменного тока [4].

Отличительной особенность системы является то, что концентратор обладает сразу двумя сетевыми интерфейсами. Для коммутации с оконечными устройствами используется технологии передачи данных ZigBee. Данный подход позволяет переводить оконечные устройства в пассивный режим ожидания, значительно снижая их энергопотребление и позволяя им работать от аккумулятора достаточно продолжительно время (до нескольких десятков месяцев). Для подключения к сети Интернет концентратор имеет модуль Wi-Fi.

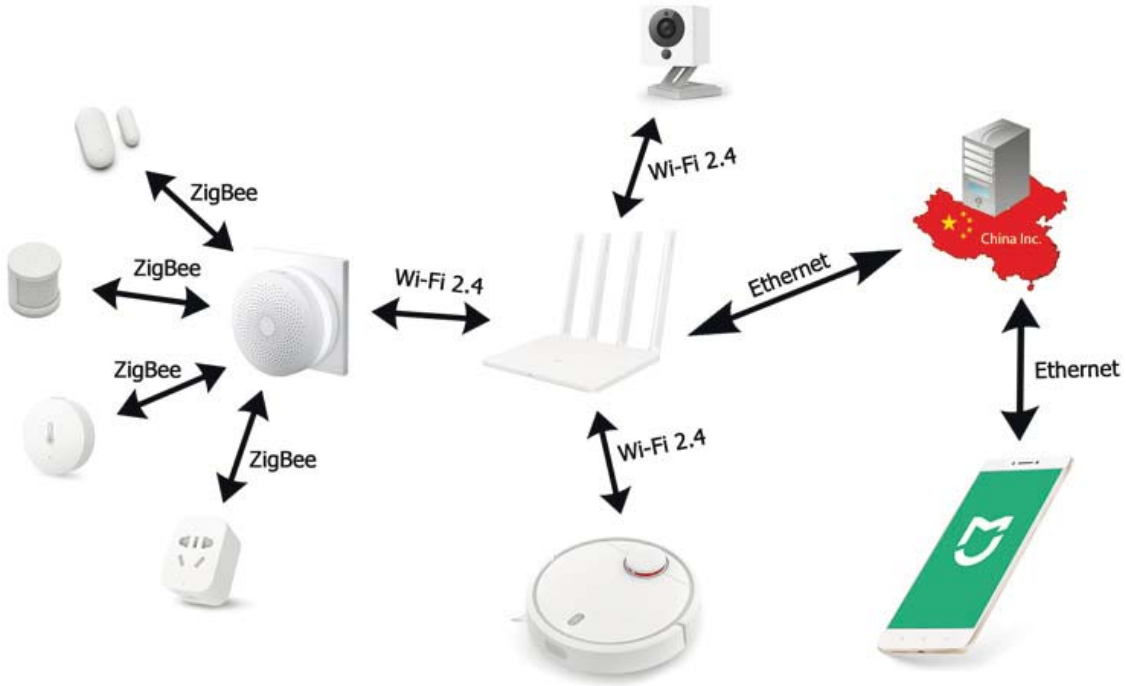

Рисунок 2 – MiJia Smart Home Set

Достоинством такой архитектуры является:

– легкая масштабируемость (так как ZigBee использует ячеистую топологию);
– простота установки и конфигурирования;
– автономность оконечных устройств;
– при отсутствии доступа к сети Интернет система локально сохраняет работоспособность в рамках сценариев, настроенных на концентраторе;
– удобное мобильное приложение;
– настройка сценариев работы;
– доставка push-уведомлений;
– относительно низкая стоимость комплекта.

К недостаткам рассматриваемой реализации можно отнести следующие:

– не все умные устройства могут использовать ZigBee для передачи данных (существуют ограничения по ширине канала передачи данных и времени распространение сигнала в случае значительно удаления оконечных устройств от концентратора при использовании нескольких ретрансляторов);
– привязка управляющих устройств к Mi аккаунту пользователя и осуществление передачи информации через сервера компании Xiaomi;
– при отсутствии доступа к сети Интернет система теряет возможности управления, конфигурирования и доставки уведомлений.

1.3. СИСТЕМА УМНОГО ДОМА VIVINT

На западных зарубежных рынках существует множество вариантов реализации систем умного дома. Одним из лидеров рынка по версии некоммерческой организации safehome является компания Vivint, которая среди своих продуктов имеет системы охранной и пожарной сигнализации.

Архитектура системы схожа с той, что была рассмотрена в пункте 1.2. В системе присутствует главное устройство – концентратор, который в данном случае совмещён с контрольной панелью, динамиком и экраном для управления устройствами [5]. Концентратор имеет два сетевых интерфейса: Wi-Fi и Z-Wave. Первый используется для подключения системы к сети Интернет, второй – для управления оконечными устройствами. Использование протокола Z-Wave вместо ZigBee первое существенное различие двух архитектур. Протоколы схожи в своей топологии и назначении, при этом разрабатываются двумя конкурирующими альянсами. Z-Wave – проприетарный протокол, в то время как ZigBee – открытый. Компания предлагает следующий набор сенсоров:

- контактные датчики открытия окон и дверей;
- датчики обнаружения движения;
- датчики разбития стекла;
- датчики обнаружения дыма и CO_2.

Для управления системами умного дома может использоваться сенсорная панель, программируемые брелоки, тревожные кнопки и кулоны, а также возможно управление голосом. Интеграция с системой умного дома Google Home является вторым существенным достоинством системы и её отличием от варианта от Xiaomi.

1.4. ВЫВОД

Анализ существующих решений показывает, что все рассмотренные системы имеют общие черты, например, во всех рассмотренных случаях системы охранной и пожарной сигнализации состояли из управляющего и оконечных устройств [6]. При этом основным недостатком традиционных кабельных систем является сложность установки, настройки и обслуживания

системы, а также отсутствие возможности интеграции с системами умного дома. Основным недостатком современных решений является их закрытость и использование сторонних сервисов компании производителя системы для хранения информации пользователя и управления системой. Предлагаемая к реализации новая система должна быть лишена этих недостатков.

2. ФОРМИРОВАНИЕ ТРЕБОВАНИЙ К СИСТЕМЕ

Реализация охранных систем и систем сигнализации накладывает на разработчиков ряд ограничений, прежде всего связанных с необходимостью обеспечения максимальной степени защиты системы от несанкционированного доступа к приватной информации и неавторизованного управления [2, 7]. Эти ограничения выдвигают ряд требований к используемым подходам инструментарию, которые необходимо учитывать при разработке таких систем.

2.1. ХРАНЕНИЕ ДАННЫХ

Исходя из специфики разрабатываемой системы и для снижения потенциальных рисков передачи персональной информации третьим лицам наилучшим вариантом является хранения данных локально, не прибегая к использованию удалённых или облачных сервисов. Данное требование накладывает ограничение на место развертывание используемых компонентов системы и определяет общую схему построения системы и специфику её работы.

2.2. АРХИТЕКТУРА СИТЕМЫ

Система предполагает модульную масштабируемую архитектуру, которая должна позволять конфигурировать систему исходя из потребностей потребителя. Базовыми в реализации являются модули полевого устройства и модуль устройства управления, который должен включать в себя следующие компоненты:

– хранение информации о структуре системы;

– рассылка сигналов управления подчиненным устройствам;
– мониторинг состояния системы и подчиненных устройств;
– обработка аварийных ситуаций (срабатывание датчиков);
– модули доставки уведомлений;
– модуль внешнего API для управления системой из вне;
– модуль голосового управления системой.

Репозиторий проекта должен содержать файл конфигурации, включающий основные настройки маршрутизации и межмодульного взаимодействия. Редактирование файла конфигурации должно позволять запустить систему с нужным количеством выбранных компонентом. Для запуска системы необходимо использовать систему сборки проекта. Идеальным вариантом является использование системы, позволяющей запускать каждый компонент в отдельном контейнере в специально созданном для него окружении. Данный подход позволит избежать конфликта используемых зависимостей и повысит степень отказоустойчивости системы в целом.

2.3. ИНФОРМАЦИОННЫЙ ОБМЕН

Разрабатываемая система должна обеспечивать совместимость с существующими распространёнными системами охранной и пожарной сигнализации, то есть иметь возможность построить новую систему на основе существующей инфраструктуры с заменого старого контроллера на новый, а также иметь возможность подключения новых беспроводных устройств, работающих в сетях Wi-Fi, Bluetooth, ZegBee, Z-Wave или других. Реализация каждого из указанных вариантов организации сети является непростой задачей, требует знаний соответствующих спецификаций, выбора программных средств для реализации подхода и наличия аппаратных средств с указанными сетевыми

интерфейсами. Далее в данной работе будет рассмотрен один из вариантов построения системы – на основе Wi-Fi сетей, то есть с использование наиболее распространённого типа сетей и, чаще всего, существующего сетевого оборудования. Использование выбранного типа сети не должно накладывать никаких ограничений на использование прочих или совместного использования нескольких из них одновременно. Требуется предусмотреть возможность дальнейшей реализации прочих типов, при сохранении работоспособности всех остальных сервисов системы.

При реализации системы на основе сетей Wi-Fi одним из способов защиты от несанкционированного доступа является использование отдельно выделенной приватной (скрытой от обнаружения) подсети для всех устройств системы, и ограничение её использования для прочих целей. Настоятельно рекомендуется периодическая проверка наличия и установка обновлений программного обеспечения маршрутизатора и установка надежных паролей.

2.3.1 ВНУТРЕННЕЕ ВЗАИМОДЕЙСТВИЕ

Для обеспечения обмена данными между устройствами в сети умного дома используются специальные протоколы. При формировании требований необходимо учитывать особенности устройств, которые используются в системе. Полный список таких требований с описанием приведён в таблице 2.

Таблица 2 – Требования к протоколу внутреннего взаимодействия

Наименование	Описание
Дуплексность	Протокол должен обеспечивать передачу сообщений управления и проверки состояния от управляющего устройства к подчиненным. С другой стороны необходимо получать от устройств информацию о их состоянии, ответы на управляющие сигналы, данные о срабатывании датчиков или возникновении технических неполадок.
Быстродействие	Подчиненные устройства в любой момент времени с минимальными затратами времени должны иметь возможность уведомить устройство верхнего уровня о возникновении аварийной ситуации или сработке датчиков. То же самое относится к обработке управляющих сигналов. Предпочтительным вариантом в данном случае являются использование протокола с установлением соединения, который, в случае, необходимости позволит обеспечить обмен информации с использованием уже созданных подключений .
Асинхронность	В силу специфики работы систем охранной и пожарной безопасности нельзя допускать возможности блокировки канала, который используется для обмена сообщениями. Любое устройство должно иметь возможность принять сообщение в момент его получения, обработать его и незамедлительно вернуть ответ в момент его готовности.
Надежность	Протокол, в силу возможностей аппаратной части, должен поддерживать режимы гарантированной доставки сообщений.
Легкая интеграция для новых устройств	Добавление новых устройств в систему должно быть реализовано “на лету”, то есть без влияния на функционирование существующей системы.

Продолжение таблицы 1

Легковесность	Требование прежде всего обосновано аппаратной реализацией полевых устройст, представленных, в общем случае, недорогими и малопотребляющими схемами, которые могут не обладать большими вычислительными возможностями для реализации сложных протоколов (например, промышленных, с большим количеством заголовочной информации в каждом пакете или с требованиями к реализации в каждом устройстве большого количества минимально требуемого набора команд).
Доступность и простота реализации	Для простоты реализации устройств выбранный протокол должен быть открытым и достаточно распространенным. В этом случае для подавляющего большинства современных контроллеров, на которых будут построены устройства, можно будет найти библиотеки базовой реализации протокола с примерами и инструкцией по их использованию, а в случае возникновения проблем обратиться за помощью к сообществу разработчиков.
Масштабируемость	При достаточной производительности управляющих устройств и сетевых контроллеров протокол не должен накладывать ограничения на размер сети и количество одновременно подключенных устройств.
Основа протокола	Для простоты интеграции и развертывания системы предполагается подключение всех устройств через существующие Wi-Fi сети. При таком подходе выбранный протокол должен иметь возможность работать на основе стека TCP/IP.

2.3.2 ВНЕШНЕЕ ВЗАИМОДЕЙСТВИЕ

Для возможности управления системой, находясь за пределами локальной сети необходимо разработать и развернуть публичное API. В данном случае применяется стандартный набор требований к такому взаимодействию:

- использование только защищённого шифрованного соединения по протоколу https, использование http не допускается;
- необходимость статического IP адреса;
- использование современных механизмов аутентификации;
- возможность отслеживания изменения состояния компонентов системы в реальном времени.

Выбранный способ реализации публичного API должен обеспечивать выше указанные требования. Кроме того, использование https протокола подразумевает получение SSL сертификатов, что в свою очередь требует наличие собственного доменного имени, а также ставит задачу настройки автоматических обновления компонентов отвечающих за безопасность внешних соединений и сертификатов.

2.4. ВЫВОД

Во второй главе были приведены основные требования к различным компонентам системы. Следующие главы, описывающие различные этапы разработки системы, будут ориентированы на реализацию этих требований.

3. ВЫБОР СРЕДСТВ РЕАЛИЗАЦИИ

Перед переходом к процессу проектирования и разработки системы необходимо решить ряд вопросов, которые могут повлиять на архитектуру будущего решения: выбрать аппаратные и программные средства для реализации системы, а также определиться с протоколом, который будет использован для организации взаимодействия устройств.

3.1. ПРОТОКОЛ ВНУТРЕННЕГО ВЗАИМОДЕЙСТВИЯ

На основании требований, приведённых в пункте 3.2.1 необходимо выбрать протокол для передачи данных в локальной сети. Наиболее распространёнными в области IoT на сегодняшний день являются следующие:

- XMPP (Extensible Messaging and Presence Protocol) [8];
- SOAP (Simple Object Access Protocol) [9];
- COAP (Constrained Application Protocol) [10];
- STOMP (Streaming Text Oriented Message Protocol) [11];
- MQTT (Message Queuing Telemetry Transport) [12].

Краткая сравнительная характеристика протоколов приведена в таблице 2.

Таблица 3 – Сравнение протоколов

Требование	XMPP	SOAP	COAP	STOMP	MQTT
Дуплексность	Да	Да	Да	Да	Да
Асинхронность	Да	Да	Да	Да	Да
Надежность	Опция	Нет	Да	Нет	Да
Легкая интеграция для	Нет	Да	Да	Да	Да

новых устройств					

Продолжение таблицы 3

Легковесность	Нет	Нет	Да	Да	Да
Доступность и простота реализации	Нет	Нет	Да	Нет	Да
Масштабируемость	Да	Да	Да	Да	Да
Основа протокола	TCP/IP/ WS	HTTP, TCP, UDP	UDP/IP	TCP/IP/ WS	TCP/IP
Централизованное управление	Да	Нет	Нет	Нет	Да

Анализ протоколов показывает, что каждый из них предназначен для двустороннего обмена сообщениями в асинхронном режиме. Все протоколы могут обеспечить требуемую производительность для наших целей. Однако, помимо достоинств, стоит выделить ряд существенных различий. Большинство различий вытекают из основы используемого протокола, то есть из того сетевого стека, что лежит в его основе. Доступность и простота реализации оценивалась исходя из теоретической возможности реализации протокола на плате уровня ESP8266. Для протоколов, которые возможны к реализации оценивалось наличие стандартных средств, библиотек и инструкций к ним.

Протокол XMPP не отличается легковесностью и требует установки довольно требовательного сервера. Все сообщения протокола используют тяжеловесный формат XML, при этом форматы этих сообщений протокола описываются несколькими десятками различных спецификаций (XEPs) и являются обязательными для реализации на клиенте для корректной работы системы. Для подключения нового устройства необходима регистрация на

XMPP сервере. Несмотря на достоинства протокола XMPP, такие как масштабируемость, возможность контроля доставки сообщений, хранение данных клиентов и истории сообщений, его использование в разрабатываемой системе неоправданно и излишне.

Протокол SOAP схож по своему назначению с протоколом HTML (чаще всего работает на его основе) и реализует его подходы при передаче сообщений. При этом, как и в предыдущем случае использует тяжеловесные сообщения формата XML.

Протокол STOMP имеет все достоинства протокола SOAP, а также не накладывает ограничений на формат сообщений. Однако также не обеспечивает гарантированной доставки сообщений уровне самого протокола.

Оставшиеся два протокола COAP и MQTT в общем случае отвечают всем выдвинутым требования. В рамках нашей задачи основным различием этих протоколов является использование централизованного сервиса для обмена сообщениями. В случае с COAP клиенты обмениваются сообщениями между собой, а в случае с MQTT для обмена сообщениями используется центральный ресурс – MQTT брокер. Достоинством первого подхода является отсутствие необходимости запуска каких-либо сервисов. Достоинством второго подхода, по сравнению с первым, является простота реализации клиентов. Возможность снизить нагрузку на оконечные устройства, наличие реализаций MQTT брокера с открытым исходным кодом, а также наличие подавляющего количества, в сравнение с прочими протоколами, стандартных реализаций клиентов и инструкций к ним определили использование MQTT протокола для реализации системы. Рассмотрим выбранный протокол более подробно.

3.1.1. ОПИСАНИЕ ПРОТОКОЛА MQTT

MQTT (Message Queue Telemetry Transport) – это легковесный, компактный и открытый протокол обмена данными, созданный для передачи данных на удаленных локациях, где требуется небольшой размер кода и есть ограничения по пропускной способности канала. Протокол изначально ориентирован на применения в системах M2M (Machine to Machine) и поэтому получил свое распространение в сфере IoT. Первая версия протокола была разработана Энди Стэнфорд-Кларком (IBM) и Арленом Ниппером (Arcom) в 1999 году и опубликована под свободной лицензией. На момент времени написания данной работы широко распространена версия протокола 3.3.1, которая в 2014 году была стандартизирована консорциумом OASIS [12], а в 2016 году получила статус стандарта ISO/IEC с номером 20922. Следует упомянуть, что апреле 2019 стала доступна новая версия протокола MQTT – 5.0, которая, однако еще не успела получить широкого распространения, поэтому в данной работе мы будем использовать версию 3.3.1.

Протокол MQTT работает на прикладном уровне поверх TCP/IP (см. рисунок 3) и использует по умолчанию 1883 порт (8883 при подключении через SSL).

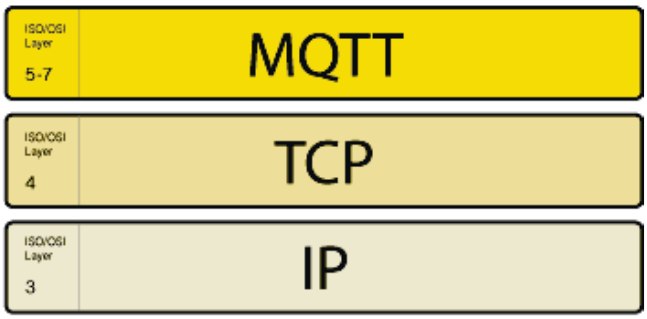

Рисунок 3 – Место протокола MQTT в модели ISO/OSI

Обмен сообщениями в протоколе MQTT осуществляется между клиентами (clients) и брокером (broker). К первым относят издателей сообщений (publishers) или подписчиков (subscribers) на сообщения. Издатели отправляют данные MQTT брокеру, указывая в сообщении определенную тему (topic). Все клиенты, подписанные на указанную тему, являются подписчиками получат это сообщения от брокера. Таким образом, с одной стороны, клиент может отправлять сообщения по любому количеству различных тем. С другой стороны, он может быть подписанным и получать сообщения по неограниченному количеству уникальных для себя тем. Функции по поддержанию соединений, доставке сообщений, их маршрутизации и обеспечению QoS ложатся на MQTT брокера, в то время как клиенты остаются легковесными и используют возможности брокера.

В MQTT определены 5 типов сообщений для взаимодействия клиентов с брокером, ниже представлены основные:

– Connect – установить соединение с брокером;
– Disconnect – разорвать соединение с брокером;
– Publish – опубликовать данные в тему;
– Subscribe – подписаться на тему;
– Unsubscribe – отписаться от темы.

Схема простого взаимодействия между подписчиком, издателем и брокером представлена на рисунке 4.

Рисунок 4 – Взаимодействие клиентов с брокером в MQTT протоколе

Темы представляют собой символы с кодировкой UTF-8. Их иерархическая структура имеет формат «дерева», что упрощает организацию и доступ к данным. Как правило, темы состоят из одного или нескольких уровней, которые разделены между собой символом «/». Пример темы в который охранный датчик, расположенный в спальной комнате публикует данные брокеру: */home/living-space/living-room1/security*. При такой древовидной структуре наименования подписчик может легко получать данные сразу с нескольких тем, для этого существуют так называемые wildcard. Они бывают двух типов: одноуровневые и многоуровневые.

Одноуровневый wildcard применяется символ «+». К примеру, нам необходимо получить данные о состоянии охранных датчиков во всех спальных комнатах. При этом в каждой комнате установлены также датчики пожарной сигнализации и устройства для управления освещением:

- */home/living-space/living-room1/security;*
- */home/living-space/living-room2/security;*
- */home/living-space/living-room1/fire;*
- */home/living-space/living-room2/fire;*
- */home/living-space/living-room1/light1;*
- */home/living-space/living-room1/light2.*

Получить данные только с устройств охранной сигнализации можно с помощью следующего варианта подписки: */home/living-space/+/security*. При этом клиент получит сообщения только из первых двух интересующих его тем и не получил сообщения из оставшихся.

Для обозначения многоуровневого wildcard применяется символ «#», который позволяет подписаться на все темы с общим родительским элементом дерева. Допустим, что помимо жилого пространства (*living-space)* в доме существуют другие зоны, например, отдельно выделяется подвал (*basement*):

– */home/basement/living-room1/security;*
– */home/basement/living-room2/security;*
– */home/basement/living-room1/fire.*

В этом случае получить список всех тем из первого списка можно с помощью подписки вида: */home/living-space/#*.

Протокол MQTT поддерживает три уровня качества обслуживания (QoS) при передаче сообщений. **QoS 0** (At most once) реализует следующую логику: издатель один раз отправляет сообщение брокеру и не ждет подтверждения от него (см. рисунок 5).

QoS 1 (At least once). Этот уровень гарантирует, что сообщение точно будет доставлено брокеру, но есть вероятность дублирования сообщений от издателя. После получения дубликата сообщения, брокер снова рассылает это сообщение подписчикам, а издателю снова отправляет подтверждение о получении сообщения. Если издатель не получил PUBACK сообщения от брокера, он повторно отправляет этот пакет.

QoS 2 (Exactly once). На этом уровне гарантируется доставка сообщений подписчику и исключается возможное дублирование отправленных сообщений. Издатель отправляет сообщение брокеру. В этом сообщении указывается уникальный Packet ID и QoS=2. Издатель хранит сообщение неподтвержденным пока не получит от брокера ответ PUBREC. Брокер отвечает сообщением PUBREC в котором содержится тот же Packet ID. После его получения издатель отправляет PUBREL с тем же Packet ID. До того, как брокер получит PUBREL он должен хранить копию сообщения у себя. После получения PUBREL он удаляет копию сообщения и отправляет издателю сообщение PUBCOMP о том, что транзакция завершена.

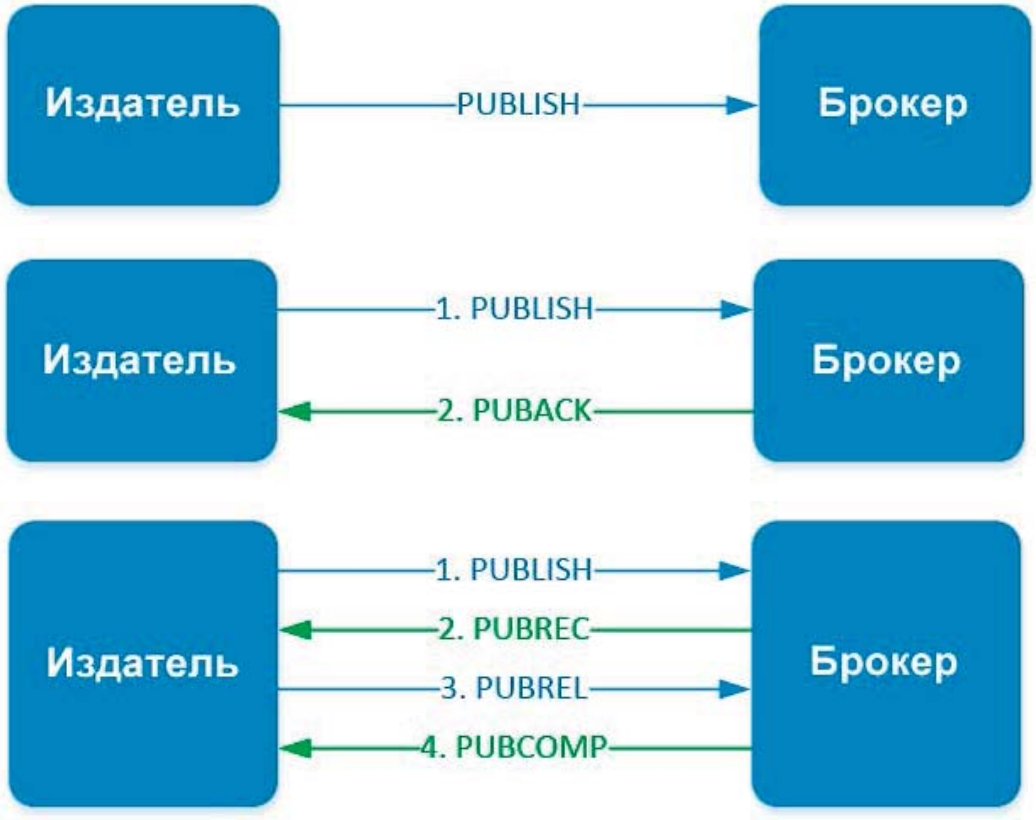

Рисунок 5 – Уровни QoS в MQTT

Для обеспечения безопасности в MQTT протоколе реализованы следующие методы защиты:

– Аутентификация клиентов. Пакет CONNECT может содержать в себе поля USERNAME и PASSWORD. При реализации брокера можно использовать эти поля для аутентификации клиента;
– Контроль доступа клиентов через Client ID;
– Подключение к брокеру через TLS/SSL.

3.2. ВЫБОР АППАРАТНЫХ СРЕДСТВ

3.2.1. ПОЛЕВЫЕ УСТРОЙСТВА

Для разработки подчиненных устройств была выбрана плата на чипе ESP32 – Wi-Fi ESP-WROOM-32. Внешний вид платы приведён на рисунке 6, а краткая спецификация в таблице 4 [13].

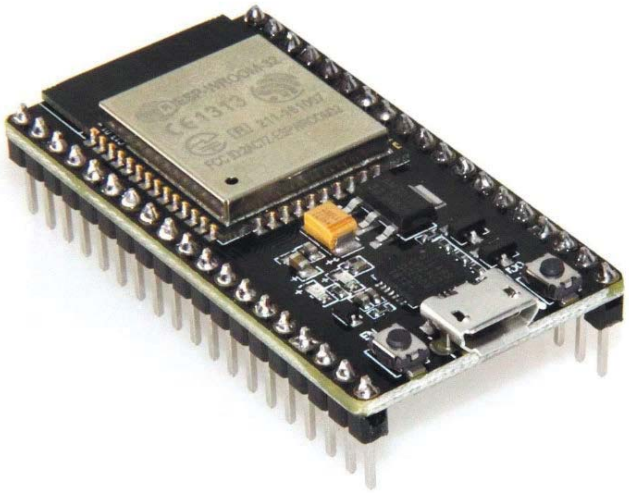

Рисунок 6 – Внешний вид платы Wi-Fi ESP-WROOM-32

Согласно спецификации, плата имеет встроенный Wi-Fi контроллер и 10 емкостных программируемых выходов, которые могут быть использованы для подключения датчиков. В данной работе для демонстрации работы модуля полевого устройства будут использовать модули датчиков движения HC-SR501, которые имеют два вывода для подключения питания и один информационный вывод для подключения к используемой плате Wi-Fi ESP-WROOM-32.

Таблица 4 – Спецификация Wi-Fi ESP-WROOM-32

Основные
CPU: Xtensa Dual-Core 32-bit LX6, 160 MHz or 240 MHz (up to 600 DMIPS) Memory: 520 KByte SRAM, 448 KByte ROM Flash: 1, 2, 4… 64 Мб
Беспроводные интерфейсы:
Wi-Fi: 802.11b/g/n/e/i, до 150 Mbps с HT40 Bluetooth: v4.2 BR/EDR and BLE
Периферия:
12-bit SAR ADC up to 18 channels 2 × 8-bit DAC 10 × touch sensors Temperature sensor 4 × SPI 2 × I²S 2 × I²C 3 × UART 1 host (SD/eMMC/SDIO) 1 slave (SDIO/SPI) Ethernet MAC с поддержкой DMA и IEEE 1588 CAN 2.0 IR (TX/RX) LED PWM up to 16 channelsHall sensor Ultra low power analog pre-amplifier
Безопасность:
IEEE 802.11 security WFA, WPA/WPA2 и WAPI Secure boot Flash encryption 1024-bit OTP, up to 768-bit for task Cryptographic engine: AES, SHA-2, RSA, ECC, RN

3.2.1. УСТРОЙСТВО УПРАВЛЕНИЯ

В качестве аппаратной платформы для управляющего устройства предлагается использовать одноплатформенный компьютер Raspberry Pi 3

версии B+ [14]. Внешний вид платы приведён на рисунке 7, а краткая спецификация в таблице 5. Плата представляет собой небольшой, но полноценный компьютер, работающий под управление операционной Linux. Для своих устройств производитель предоставляет свою версию операционной системы – Raspbian [15] построенную на базе дистрибутива Debian. Последнее обеспечивает лёгкость переноса и совместимость всех наработок с рабочего компьютера, работающего под операционной системой, основанной на образе Debian, на устройство.

Raspbian поставляется в нескольких конфигурациях. Наиболее подходящей в нашем случае является Raspbian Stretch Lite которая построена на базе Debian Stretch. Данная операционная система имеет минимально необходимое количество предустановленных пакетов, и по умолчанию не включает графическую оболочку. Управление можно осуществлять через консоль при непосредственном подключении клавиатуры и монитора к плате или через удаленное подключении к плате по ssh. Для удобства использования второго варианта взаимодействия в настройках маршрутизатора необходимо зарезервировать для Raspberry Pi статический адрес в локальной сети.

Рисунок 7 – Внешний вид платы Raspberry Pi версии B+

Таблица 5 – Спецификация платы Raspberry Pi версии B+

Основные
Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC @ 1.4GHz 1GB LPDDR2 SDRAM
Беспроводные интерфейсы:
2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN Bluetooth 4.2, BLE Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps)
Периферия:
Extended 40-pin GPIO header Full-size HDMI 4 x USB 2.0 ports CSI camera port for connecting a Raspberry Pi camera DSI display port for connecting a Raspberry Pi touchscreen display 4-pole stereo output and composite video port Micro SD port for loading your operating system and storing data 5V/2.5A DC power input Power-over-Ethernet (PoE) support (requires separate PoE HAT)

Использование одноплатформенного компьютера не накладывает существенных ограничений на выбор используемых аппаратных средств. Использование операционной системы Linux позволяет развертывать различные сервисы и использовать любые средства разработки, однако, в силу ограниченности используемой платформы по производительности процессора, объему оперативной и постоянной памяти, приоритетными являются легковесные платформы, не требующие установки большого количества компонентов окружения.

3.3. ВЫВОД

В третьей главе были рассмотрены основные протоколы, используемые в области IoT. На основе их анализа и сравнения был выбран протокол MQTT, как наиболее полно удовлетворяющий выдвинутым ранее требованиям. Были выбраны программные и аппаратные средства для реализации будущей системы.

4. РАЗРАБОТКА АРХИТЕКТУРЫ

Как уже говорилось в п.2.2 система в простейшем исполнении состоит из управляющего устройства и набора полевых устройств. Также там был приведён список основных модулей управляющего устройства.

В данной главе рассматривается разработка схемы взаимодействия управляющего и управляемых устройства и схема взаимодействия основных компонентов управляющего устройства, включая систему доставки уведомлений пользователю.

4.1. ВЗАИМОДЕЙСТВИЕ УСТРОЙСТВ

Общий вид схемы подключения устройств приведён на рисунке 8. MQTT брокер разворачивается в отдельном контейнере в составе управляющего устройства. Для подключения к брокеру и устройство управления и полевые устройства запускают собственные MQTT клиент.

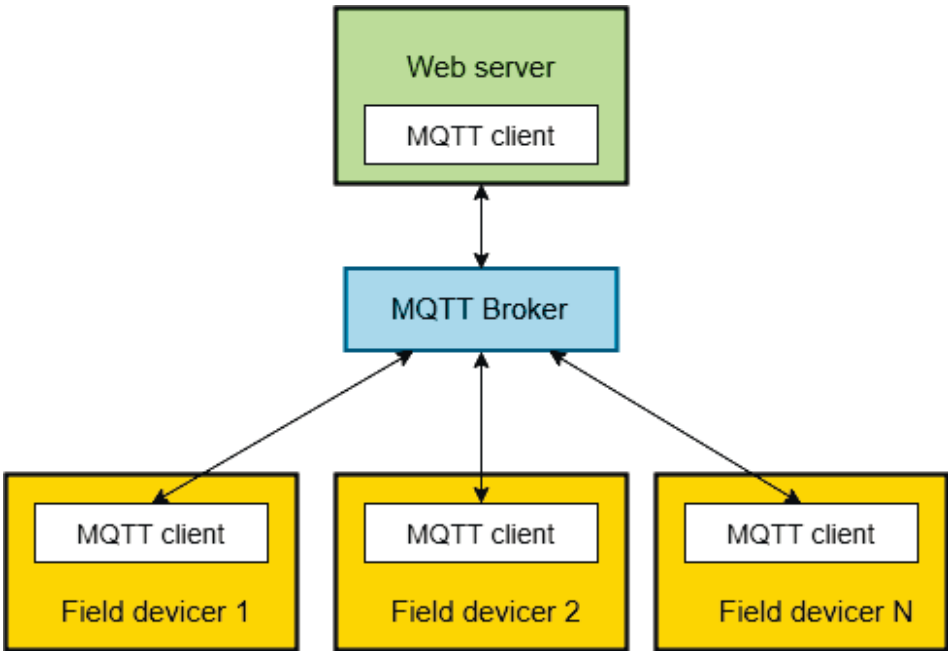

Рисунок 8 – Взаимодействие устройств в локальной сети

Всё взаимодействие осуществляется посредством подписок и публикаций сообщений в определённые темам, как было показано в п. 3.3.1. Каждое полевое устройство реализует следующую логику работы (см. рисунок 9):

1. Настраивает выводы для подключения датчиков на приём сигналов.
2. Выполняет подключение к сети по Wi-Fi.
3. Выполняет подключение к MQTT брокеру.
4. Хранит в памяти свой уникальный идентификатор *(deviceId)*, тип устройства *(deviceType)* и текущее состояние устройства (*deviceStatus*): включено, выключено или сработка сигнализации.
5. Отображает свое состояние с помощью внешнего светодиода.
6. После установления соединения с брокером устройство устанавливает свой статус в состояние выключено и отправляет тип устройства в тему /*auth/deviceId*.
7. Для получения сигналов управления устройство подписывается на тему */control/deviceId*.
8. При получении сообщения из темы /control/deviceId устройство устанавливает полученный статус в качестве своего текущего (доступны два значения: 0 – выключено, 1 – включено, прочие значения игнорируются), после смены статуса устройство отправляет своё состояние в тему /state/deviceId.
9. Для осуществления контроля работоспособности устройство подписывается на тему */ping*, а при получении сообщения в ней отправляет ответ свой идентификатор *deviceId* в тему */pong*.
10. Полевое устройство непрерывно следит за состоянием подключенных датчиков и при их срабатывании (возникновение аварийной ситуации)

проверяет текущий статус устройства. В случае если система включена, отправляет сообщение со своим идентификатором в тему */alarm /deviceId*.

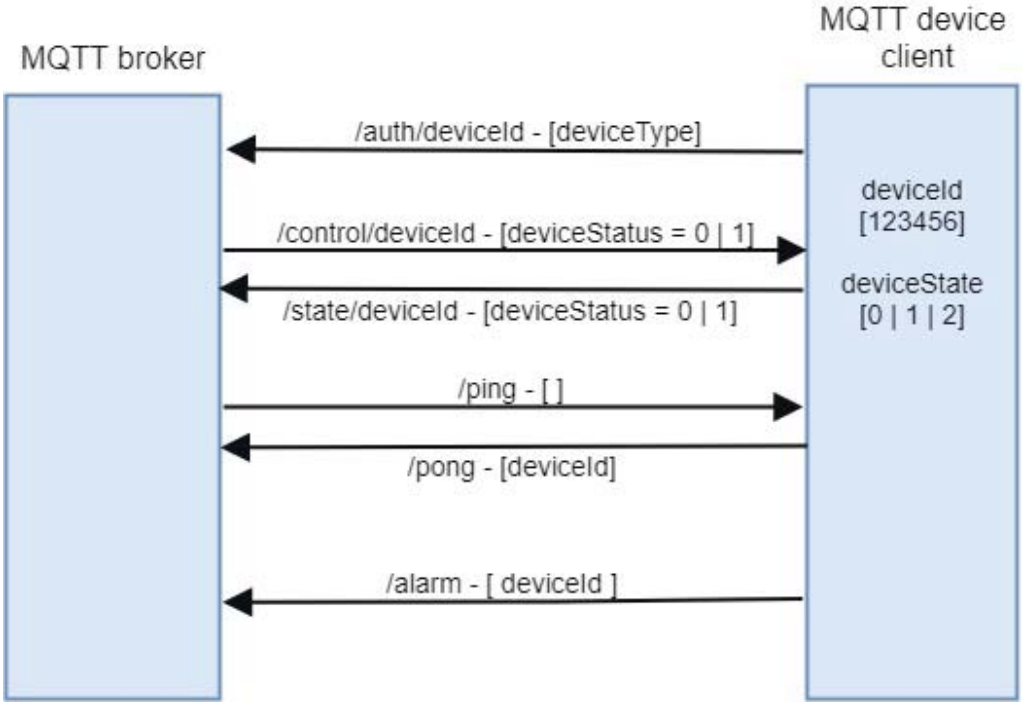

Рисунок 9 – Взаимодействие полевых устройств с MQTT брокером

Управляющее устройство в свою очередь реализует следующую логику взаимодействия с MQTT брокером (см. рисунок 10):

1. Кеширует данные текущей конфигурации системы: список всех подключенных устройств и их идентификаторов, состояние и тип каждого устройства, при обновлении данных сохраняет их в постоянной памяти.
2. Подписывается на тему */auth/+* и обрабатывает сообщения от вновь подключенных устройств – сохраняет их идентификаторы и типы в общий список устройств и добавляет их в группу нераспределённых устройств.
3. Периодически рассылает контрольное сообщение в тему */ping*, а для контроля работоспособности устройств подписывается на сообщения из

темы */pong* и составляет список всех ответивших устройств, сопоставляя списки выполняет проверку работоспособности подключенных устройств.

4. При внешнем вызове осуществляет включения и отключение охраны для указанной группы, для чего управляющее устройство рассылает сообщения для списка устройства группы по темам */control/deviceId*, а для осуществления контроля установки состояния подписывается на соответствующие темы */state/deviceId*.
5. Подписывается на тему */alarm* для получения сообщений о сработке системы.

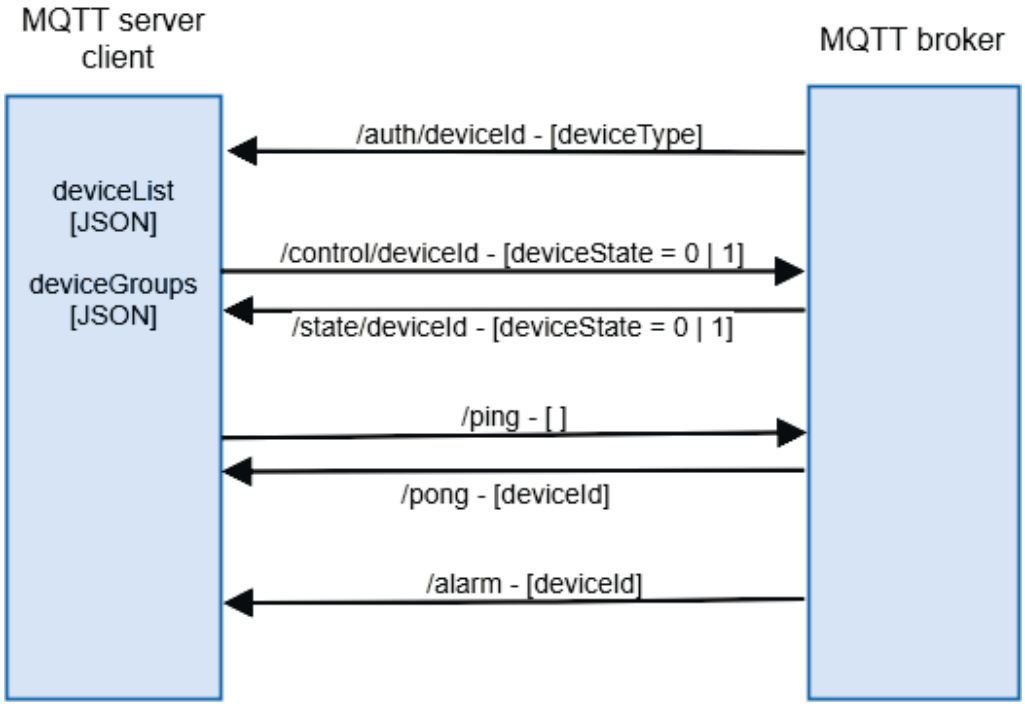

Рисунок 10 – Взаимодействие управляющего устройства с MQTT брокером

4.2. ВНЕШНЕЕ УПРАВЛЕНИЕ

Система должна предоставлять публичное API для внешнего управления. В частности, возможность получения информации о системе, например, для от рисовки интерфейса пользователя со списком групп устройств и их статусов в

мобильном приложении. Кроме того, необходимы инструменты для управления состоянием системы – постановки и снятия с охраны указанной группы устройств. Для настройки системы должны быть реализованы методы добавления устройства в указанную группу. Ниже приведён список необходимых методов с описанием принципов их работы и форматами сообщений. Все методы используют формат JSON для формирования тел запроса и ответа. Все рассмотренные ниже методы возвращаю 200 код в случае успешного выполнения запроса.

1. Метод получение информации о системе – GET */group* – служит для запроса списка групп устройств, их статуса и списка входящих в группу устройств. Тело запроса не требуется. Тело ответа имеет формат, представленный на рисунке 11.
2. Метод создания новых групп – POST */group*. В теле запроса принимает название группы, а в теле ответа возвращает идентификатор созданной группы.
3. Метод удаления групп – DELETE */group*. В теле запроса принимает идентификатор удаляемой группы. Тело ответа не требуется.
4. Метод постановки группы на охрану – POST */group/on* – устанавливает состояние «на охране» всех устройств указанной группы. Тело запроса содержит идентификатор группы. Тело ответа содержит информацию о указанной группе и её устройствах.
5. Метод снятия группы с охраны – POST */group/off* – устанавливает состояние «на охране» всех устройств указанной группы. Тела запроса и ответа аналогичны методу, описанному в предыдущем пункте.
6. Метод добавления устройств в группу – POST */group/devices* служит для добавления оконечного устройства в новую группу. Тело запроса

содержит идентификаторы группы и устройства. Тело ответа не требуется.

7. Метод добавления устройств в группу – DELETE */group/devices* служит для добавления оконечного устройства в новую группу.


```
{
  "groups": [
    {
      "id": 0,
      "name": "Новые устройства",
      "state": 0,
      "device": [
        {"id": 555551, "type": 2},
        {"id": 555552, "type": 3},
        {"id": 555553, "type": 4}
      ]
    },
    {
      "id": 1,
      "name": "Пожарная сигнализация - 1 этаж",
      "state": 0,
      "device": [
        {"id": 111111, "type": 1},
        {"id": 111112, "type": 1}
      ]
    },
    {
      "id": 2,
      "name": "Охранная сигнализация - гараж",
      "state": 1,
      "device": [
        {"id": 111201, "type": 5},
        {"id": 111202, "type": 6},
        {"id": 111203, "type": 7}
      ]
    }
  ],
  "time": "25-05-2019 22:24:36"
}
```


Рисунок 11 – Формат тела ответа внешнего API

Все описанные выше методы должны включать валидацию входных данных и возвращать корректные сообщения с описанием ошибки в случае их невалидности.

Схема реализации публичного API является классической, однако согласно выдвинутым требованиям, необходимо обеспечить передачу всей информации только через защищенное соединение https. Архитектурно это проще всего реализовать, используя проксирующий сервер, как единую точку доступа для всех внешних соединений (см. рисунок 12).

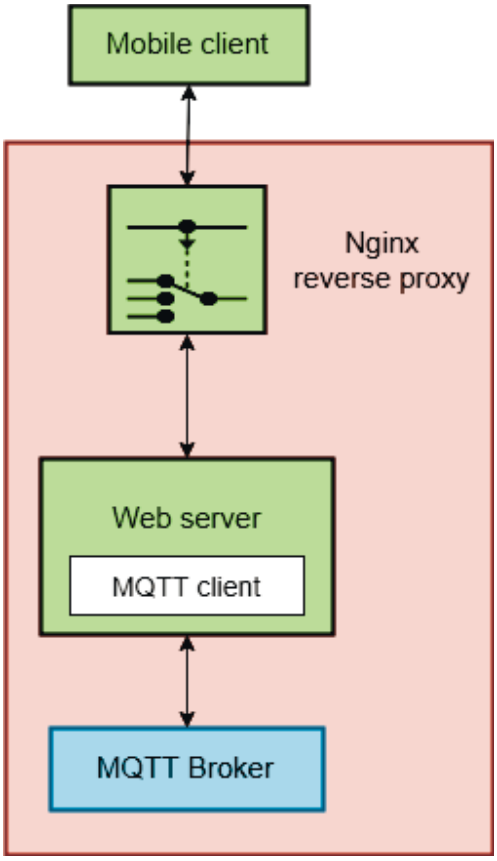

Рисунок 12 – Использование reverse proxy сервера

4.3. ДОСТАВКА УВЕДОМЛЕНИЙ

Одной из основных функций системы охранной и пожарной сигнализации является предупреждение пользователей о возникновении опасной ситуации. Для реализации этих функций необходимо разработать модуль доставки уведомлений. Реализация должна поддерживать развитие системы доставки

уведомлений и позволять легко интегрировать новые подмодули. Сложность состоит в сопряжении подмодулей модулей, которые реализуют разные способы доставки уведомлений, в частности используют разные каналы связи с разной пропускной способность и временем доставки. В представленной ниже реализации для их объединения используется механизм очередей. Схематично схема организации такой системы представлена на рисунке 13.

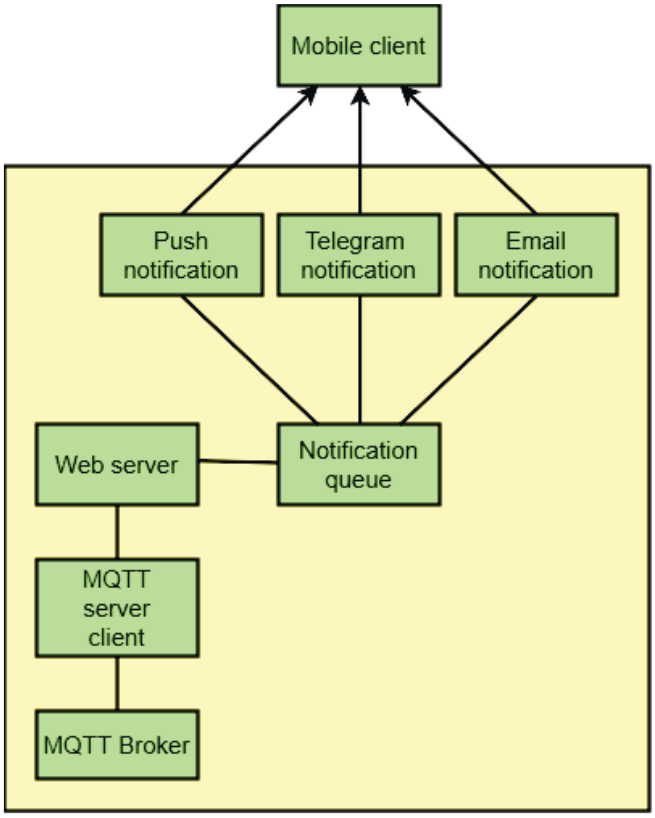

Рисунок 13 – Система доставки уведомлений

В случае возникновения ситуации, когда система должна уведомить пользователя о возникновении какого-то события веб-сервер формирует сообщение, получает из настроек пользователя, те типы доставки уведомления, которые он указал и складывает задачи на доставку уведомлений в соответствующие очереди. С каждой очередью связаны один и более обработчиков – отдельных модулей, которые выполняют доставку уведомлений

определённого типа. Каждый обработчик разворачиваются в отдельном контейнере является независимым от других.

Базовыми сценариями работы системы, требующими уведомления пользователя, являются:

- обнаружение подключения нового устройства;
- обнаружение отключения устройства (отсутствие ответа на сообщение *«/ping»*;
- обнаружение сработки устройства;
- взятие группы устройств на охрану;
- снятие группы устройств с охраны.

4.3. ВЫВОД

В четвертой главе были подробно рассмотрены основные компоненты системы и способы их взаимодействие друг с другом. Полная схема архитектуры системы приведена в листинге А.1 приложения А.

5. РЕАЛИЗАЦИЯ

В данной главе будет собрана информация о выбранных и используемых программных компонентах, и средствах разработки, а также будут рассмотрены особенности и сложности реализации отдельных компонентов в рамках разработанной ранее архитектуры.

5.1. MQTT БРОКЕР

Первым шагом реализации всей системы является запуск MQTT брокера. Существует несколько решений с открытым исходным кодом выполняющих функции брокера MQTT сообщений. Одним из самым распространённых является Eclipse Mosquitto [16]. Eclipse Mosquitto полностью поддерживает требуемую нам версию протокола 3.3.1 и может быть развернут в docker контейнере из соответствующего официального образа доступного на Docker Hub [17].

5.2. РАЗВЕРТЫВАНИЕ КОМПОНЕНТОВ

Наряду с MQTT брокером все прочие компоненты системы также разворачиваются в отдельных контейнерах. Реализуется это с помощью платформы Docker [18] и конфигурационного файла *docker-compose.yml* [19], который позволяет запускать сервисы на основе загружаемых из официальной библиотеки Docker Hub образов, либо собирать свои с использованием исполняемых исходных кодов. Сборку и запуск собственных компонентов проще всего осуществлять на основе существующих образов. В данной работе в качестве таких использовались образы платформы Node.js версии 12.4 [20] и

Python 3.7 [21]. Содержание конфигурационного файла для запуска разрабатываемой системы приводится в листинге Б.1 приложения Б.

5.3. ПОЛЕВЫЕ УСТРОЙСТВА

Разработка модуля полевого устройства осуществляется на языке программирования C. Для управления компонентами платы, в частности сетевым Wi-Fi контроллером, используются библиотеки ядра, поставляемые производителем платы ESP 32 – компанией Espressif Systems [13]. В качестве MQTT клиента используется библиотека PubSubClien [22]. Для написания и редактирования кода используется среда разработки Arduino IDE. Пример код модуля полевого устройства приведён в листинге В.1 приложения В.

При запуске устройства в методе *void setup( )* последовательно выполняется настройка контактов для подключения сенсоров, подключение к указанной Wi-Fi сети и установление соединения с MQTT брокером. Основная логика работы устройства содержится в методе *void loop( )*, где обрабатываются все поступающие по протоколу MQTT сообщения. В методе *callback(char* topic, uint8_t* payload, uint length)* реализуется логика, описанная ранее в п. 4.1. Отслеживание срабатываний сенсоров, подключенных к устройству осуществляется через механизм прерываний.

5.4. УСТРОЙСТВО УПРАВЛЕНИЯ

Ядро устройства управления представляет собой классический веб-сервер развернутый на платформе Node.js [23] с использованием фреймворка Express [24] на языке программирования Java Script. Для взаимодействия с MQTT брокером используется распространяемая через пакетный менеджер npm

библиотека mqtt [25]. Логика работы описанная в п.4.1 для управляющего устройства реализуется в обработчике получения сообщений от MQTT брокера *mqttClient.on('message', (topic, message) => { })*. Для хранения состояния устройств системы используется библиотека «node-persist» [26], которая предоставляется инструменты для работы с хранилищем данных. Для взаимодействия с очередью доставки уведомлений используется библиотека «ampqlib» [27]. Описанные в п.4.2 методы реализуются стандартными средствами фреймворка Express. Содержащий бизнес логику код ядра управляющего устройства приведён в листинге Г.1 приложения Г.

5.4.1. ПРОКСИ-СЕРВЕР

Отдельного рассмотрения требует процесс запуска и настройки прокси-сервера, так как использование протокола https требует выполнения некоторого количества действий от разработчика.

Для использования https необходимо получить SSL сертификат. Сделать это можно бесплатно, например, с помощью открытого сервиса сертификации letsencrypt [28]. Самый простой способ – воспользоваться сервисом для автоматизации процесса запуска сервера с необходимыми сертификатами. Одним из таких является сервис certbot [29], в документации к которому приводятся подробные инструкции для подавляющего большинства современных веб-сервисов и операционных систем. В данной работе использован сервер nginx [30], который будет служить единой точкой доступа к системе извне, проксируя все запросы и перенаправляя их на требуемые сервисы.

Выбранный способ получения SSL сертификатов не возможет для IP адресов, поэтому нам необходимо получить собственное доменное имя. В виду

использования его только в текущей системе с единственной целью можно выбрать имя из любой доменной зоны. В данной реализации доменное имя было зарегистрировано бесплатно в доменной зоне «.tk» с помощью сервиса freenom [31]. Регистрация доменного имени предполагает наличие статического белого IP адреса. Это не является существенным ограничением, так как большинство провайдеров предоставляют такую услуга за небольшую абонентку плату.

5.4.2. СИСТЕМА ДОСТАВКИ УВЕДОМЛЕНИЙ

Каждый обработчик разворачиваются в отдельном контейнере является независимым от других. Для реализации механизма очередей используется брокер с открытым исходным кодом RabbitMQ [32], который развертывается с помощью официального образа, доступного доступен на hub.docker.com [33].

В данной работе реализованы три варианта уведомления пользователя: с помощью data push, отправки email и сообщений в Telegram. Все 3 компонента написана на Python 3, а для подключения к брокеру используют библиотеку pika [34]. Выбор языка в данном случае обусловлен доступность необходимого инструментария.

Для доставки push уведомлений используется сервис Firebase Cloud Messaging [35]. Сторона серверной части реализация отправки push уведомлений проста и требует от разработчика регистрации в консоли разработчика Google, создания проекта и загрузки файла настроек «google-service.json». Файл содержит большое количество информации, в том числе специальный токен для отправки push уведомлений, а сама отправка уведомления представляет собой запрос к специальному ресурсу с полученным токеном и информацией, которую необходимо передать пользователю. Пример

реализации обработчика рассылки push уведомлений приведён в листинге Д.1 приложения Д.

Для отправки email используются компоненты пакета «email.mime» из стандартной поставки библиотек для языка Python. От разработчика требуется создать объект отправляемого письма, корректно указать поле адреса доставки и сформировать контент используя различные поставляемые типы. Отправка письма производится от лица указанного пользователя. Для корректной работы компонента в настройка аккаунта, который используется для рассылки писем необходимо разрешить небезопасным приложениям доступ к аккаунту. Пример реализации компонента приведён в листинге Е.1 приложения Е.

Отправка уведомлений в Telegram реализована через механизм ботов. Для реализации компонента необходимо зарегистрировать своего бота и получить его уникальный токен. Далее использую его можно осуществлять управление ботом. Система API Telegram довольна объёмна и сложна, поэтому для упрощения процесса разработки используют специальные SDK. В данной работе для реализации компонента использовалась библиотека pyTelegramBotAPI [36], которая позволяет создать объект бота и использовать его методы для отправки сообщений пользователям. Пример реализации приведён в листинге Ж.1 приложения Ж.

6. РАЗВИТИЕ СИСТЕМЫ

Одним из основных требований к системе было обеспечение её расширяемости. В рамках рассматриваемой работы был создан прототип системы для демонстрации её работы и тестирования. В дальнейшем система может быть развита путём разработки и включения новых модулей. Ниже приведён список задач, рассматриваемых для реализации в будущем с приблизительной оценкой их сложности (от 1 до 10, где 1 – минимальная сложность, а 10 – максимальная, требующая знаний о работе системы в целом и дополнительного изучения какой-либо специфической тематики). Для некоторых задач приведено краткое описание:

1. Добавить устройства световой и звуковой индикации и сигнализации (2).
2. Добавление новых оконечных устройств в существующую реализацию, основанную на сетях Wi-Fi (3).
3. Добавить модуль доставку уведомлений через любой другой канал, например, через голосовых помощников (по словам представителей компании Yandex с последних конференций разработчиков, планируется выпуск SDK для разработки систем умного дома с Алисой) (4-8).
4. Добавить возможность начального конфигурирования оконечных устройств (посредством отслеживания на смартфоне Wi-Fi сетей с зарезервированными именами, создаваемыми подключаемыми полевыми устройствами, подключения к ним и отправки начальных настроек – ssid, password, IP адрес устройства управления) (6-7);
5. Доработать интерфейс пользователя мобильного приложения с целью увеличения гибкости настроек и управления (например, добавить авторизацию, пользователей и персонализацию, возможности передачи части разрешений на управление и т.д.) (4-8).

6. Добавить экран для управляющего устройства (для разных одноплатформенных компьютеров существуют множество вариантов подключения дисплеев, необходимо реализовать аппаратную часть с подключением, а также разработать и реализовать интерфейс пользователя для управления и настройки системы (8).
7. Добавить доставку уведомлений через sms или звонки, например, используя плату Orange Pi 4G IoT (предлагаемая плата имеет разъём для сим-карты, поэтому её использование в качестве управляющего устройства позволит реализовать функции отправки и получения sms сообщений или выполнения и приема звонков для отправки уведомлений и приёма сигналов управления соответственно) (9).
8. Разработать свой сервис доставки push уведомлений – рассмотреть возможность реализации такого сервиса (например, начать можно с ознакомления с проектом http://airnotifier.github.io/ или его аналогами) (9).
9. Добавить интерфейсные модули Z-Wave или ZigBee для управляющего устройства (необходимо добавить в управляющее возможность взаимодействовать с полевыми устройствами, которые работают в соответствующих сетях, реализовать можно в виде модулей расширения используемой платы или изначально выбрать другую плату, где на ряду с Wi-Fi и Ethernet присутствует аппаратный модуль требуемого сетевого интерфейса) (10).
10. Создать оконечные устройства для новых типов сетей, поддержка которых будет добавлена в управляющее устройство (4-6).

ЗАКЛЮЧЕНИЕ

В представленной работе были разобраны основные аспекты разработки системы умного дома. Была рассмотрены требования, предъявляемые к данным системам, в частности к системам охранной и пожарной сигнализации. В ходе анализа существующих решению выяснилось, что современные системы могу иметь потенциальные проблемы с безопасностью при обращении персональных данных владельцев, так как они изначально ориентированы на широкое использование различных сторонних сервисов компаний производителей.

Была разработана модульная и масштабируемая архитектура системы, удовлетворяющая всем указанным требованиям и произведён выбор средств для реализации предложенного решения.

На заключительном этапе был создан прототип системы в котором были реализованы её основные компоненты. Текущая реализация имеет открытый исходный код и использует только соответствующие сторонние компоненты. Полученное решение может быть легко сконфигурировано, доработано и расширено под новые специфичные требования.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ganchev, I. Autonomous Control for a Reliable Internet of Services / I. Ganchev, R. D. Mei, H. Berg // Springer Open – 2018. – 416.
2. Celik, B. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities / B. Celik, E. Fernandes, E. Pauley // ArXiv – 2018. – 29.
3. Bolid, система охранной сигнализации с использованием автономных приборов «Орион» – https://bolid.ru/projects/iso-orion/os/#descr0. Дата обращения: 20.05.2019.
4. Xiaomi Smart Home Gateway 2 – шлюз управления умным домом – https://xiaomi-smarthome.ru/gateway-2. Дата обращения: 20.05.2019.
5. Vivint Sensors, Detectors and Miscellaneous Equipment – https://securitybaron.com/system-reviews/vivint/sensors-detectors-equipment. Дата обращения: 20.05.2019.
6. Patel, V.A. Integration of IoT and Cloud Computing and its issues: A Survey / V.A. Patel, V.K. Patel, G. Panchal // International Journal of Engineering Technology Science and Research. – October 2017. – V. 4 – I. 10.
7. Bako, A. Cyber and Physical Security Vulnerability Assessment for IoT-Based Smart Homes / A. Bako, A.I. Awad // Sensors-Basel. – March 2018. – V. 18 – I. 3.
8. The XMPP Standards Foundation develops extensions to XMPP in its XEP series – https://xmpp.org/extensions. Дата обращения: 21.05.2019.
9. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C Recommendation 27 April 2007 – https://www.w3.org/TR/soap12-part1. Дата обращения: 21.05.2019.

10. RFS 7252 The Constrained Application Protocol (CoAP), June 2014 – https://tools.ietf.org/html/rfc7252. Дата обращения: 21.05.2019.
11. STOMP Protocol Specification, Version 1.2 – https://stomp.github.io/stomp-specification-1.2.html. Дата обращения: 21.05.2019.
12. MQTT Version 3.1.1. OASIS Standard, 29 October 2014. – http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Дата обращения: 21.05.2019.
13. Espressif Systems Arduino core for ESP32 – https://github.com/espressif/arduino-esp32. Дата обращения: 22.05.2019;
14. Raspberry Pi 3 Model B+. – https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus. Дата обращения: 22.05.2019.
15. Raspbian – the Foundation's official supported operating system. – https://www.raspberrypi.org/downloads/raspbian. Дата обращения: 22.05.2019.
16. Eclipse Mosquitto An open source MQTT broker – https://mosquitto.org. Дата обращения: 23.05.2019.
17. Eclipse Mosquitto Docker Official Images, version 1.6.2 – https://hub.docker.com/_/eclipse-mosquitto. Дата обращения: 22.05.2019.
18. Docker – Enterprise Container Platform for High-Velocity Innovation – https://www.docker.com. Дата обращения: 22.05.2019.
19. Docker Compose is a tool for defining and running multi-container Docker applications. – https://docs.docker.com/compose. Дата обращения: 22.05.2019.
20. Node.js Docker Official Image – https://hub.docker.com/_/node. Дата обращения: 22.05.2019.
21. Python 3 Docker Official Image – https://hub.docker.com/_/python. Дата обращения: 22.05.2019.

22. PubSubClient – Arduino Client for MQTT, version 2.7 – https://github.com/knolleary/pubsubclient. Дата обращения: 22.05.2019.
23. Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine – https://nodejs.org/en. Дата обращения: 23.05.2019.
24. Express – Fast, unopinionated, minimalist web framework for Node.js – https://expressjs.com. Дата обращения: 23.05.2019.
25. MQTT.js is a client library for the MQTT protocol, written in JavaScript for node.js and the browser – https://www.npmjs.com/package/mqtt. Дата обращения: 23.05.2019.
26. Asynchronous persistent data storage on the server – https://www.npmjs.com/package/node-persist. Дата обращения: 23.05.2019.
27. AMQP 0-9-1 library and client for Node.JS – https://www.npmjs.com/package/amqplib. Дата обращения: 23.05.2019.
28. Let’s Encrypt is a free, automated, and open Certificate Authority – https://letsencrypt.org. Дата обращения: 23.05.2019.
29. Automatically enable HTTPS on your website with EFF's Certbot, deploying Let's Encrypt certificates – https://certbot.eff.org. Дата обращения: 23.05.2019.
30. NGINX Reverse Proxy – https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy. Дата обращения: 23.05.2019.
31. Freenom – Free domain provider – https://www.freenom.com/en/index.html?lang=en. Дата обращения: 23.05.2019.
32. RabbitMQ is the most widely deployed open source message broker – https://www.rabbitmq.com. Дата обращения: 24.05.2019.
33. RabbitMQ Docker Official Image – https://hub.docker.com/_/rabbitmq. Дата обращения: 24.05.2019.

34. Pika – RabbitMQ (AMQP 0-9-1) client library for Python – https://pypi.org/project/pika. Дата обращения: 24.05.2019.
35. Firebase Cloud Messaging (FCM) is a cross-platform messaging solution that lets you reliably deliver messages at no cost – https://firebase.google.com/docs/cloud-messaging. Дата обращения: 25.05.2019.
36. A simple, but extensible Python implementation for the Telegram Bot API – https://github.com/eternnoir/pyTelegramBotAPI. Дата обращения: 25.05.2019.

ПРИЛОЖЕНИЕ А
ОБЩАЯ СХЕМА АРХИТЕКТУРЫ СИСТЕМЫ

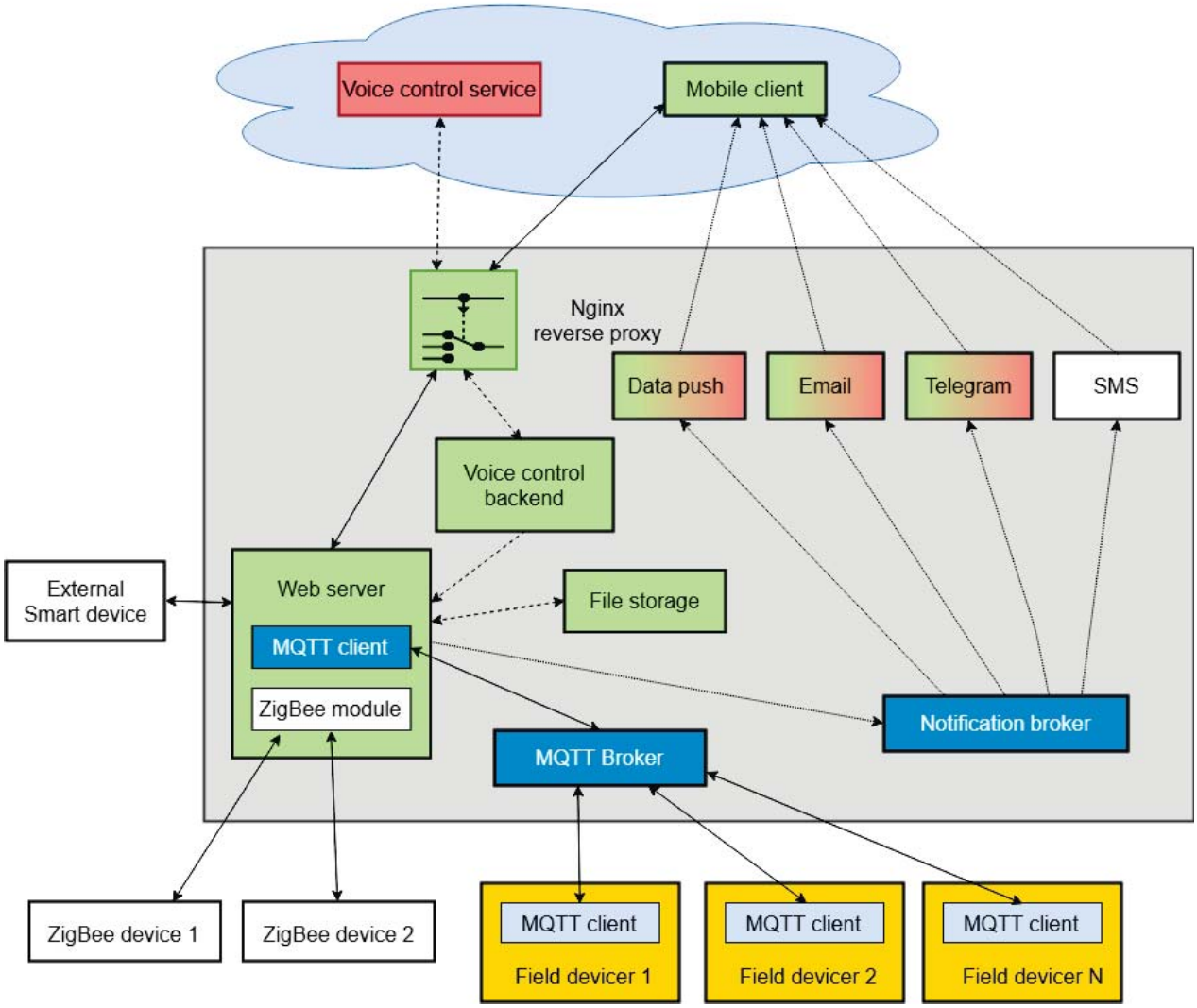

Рисунок А.1 – Общая схема архитектуры системы

ПРИЛОЖЕНИЕ Б
КОНФИГУРАЦИОННЫЙ ФАЙЛ СИСТЕМЫ

Листинг Б.1 – файл «docker-compose.yml»

```
services:

  mqtt-broker:
    image: eclipse-mosquitto:1.6
    ports:
      - 1883:1883

  rabbitmq:

    image: rabbitmq
    ports:
      - 5672:5672

  core:
    build: ./core/
    depends_on:
      - rabbitmq
      - mqtt-broker
    ports:
      - 3000:3000
    volumes:
      - ./core/test.js:/home/app/test.js
      - ./core/bin/:/home/app/bin/
      - ./core/routes/:/home/app/routes/
      - ./core/app.js:/home/app/app.js
      - ./core/logger.js:/home/app/logger.js
    command: node bin/www

  push_sender:
    build: docker/push
    depends_on:
      - rabbitmq
    volumes:
      - ./configs:/app/configs

  email_sender:
    build: docker/email
    depends_on:
      - rabbitmq
    volumes:
```

```
    - ./configs:/app/configs
```

продолжение приложения Б

```
- ./helpers/images:/app/helpers/images

telegram_sender:
  build: docker/telegram/sender
  depends_on:
    - rabbitmq
  volumes:
    - ./configs:/app/configs
    - ./docker/telegram/bot:/app/docker/telegram/bot
    - ./helpers/images:/app/helpers/images

telegram_main:
  build: docker/telegram/main
  volumes:
    - ./configs:/app/configs
    - ./docker/telegram/bot:/app/docker/telegram/bot
```

ПРИЛОЖЕНИЕ В
МОДУЛЬ ПОЛЕВОГО УСТРОЙСТВА

Листинг В.1 – файл «main.cpp»

```
#include <WiFi.h>
#include <PubSubClient.h>

const char *ssid = "name";
const char *password = "password";

const char *mqtt_server = "192.168.1.200";
const int mqttPort = 1883;

// 0 - disabled
// 1 - enabled
// 2 - device alarm
// 3 - sensor alarm
int deviceState = 0;

// 55 - Registation device type
// 66 - Check value device type
int deviceType = 55;

String deviceId = "111";
String authTopic = "/auth/" + deviceId;;
String pingTopic = "/ping";
String pongTopic = "/pong";
String controlTopic = "/control/" + deviceId;
String stateTopic = "/state/" + deviceId;
String alarmTopic = "/alarm";

// Addition pin can be added for more connected sensors
// any one from 2, 4, 18, 19, 21, 22, 23, 13, 12, 27, 26,
// 25, 33, 32, 34, 35, 36, 39 can be used for this
int pinDeviceState = 32;
int pinAlarmState = 33;
bool pinAlarmStateStatus = false;
int pin1 = 25;

WiFiClient espClient;
PubSubClient mqttClient(espClient);

void setup_init()
{
  Serial.begin(115200);
}
```

```
{
  void setup_wifi()

  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("...");
  }

  Serial.println("WiFi connected!");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void IRAM_ATTR isr() {
  if(deviceState != 3) {
    deviceState = 3;
    Serial.println("-----------------------------");
    Serial.println("Sensor triggered");
  }
}

void enable_security() {
  deviceState = 1;
  Serial.println("device enabled.");
  attachInterrupt(pin1, isr, RISING);
  Serial.println("Pin attached.");
  mqttClient.publish((char*) stateTopic.c_str(), (char*) String(deviceState).c_str());
}

void disable_security() {
  deviceState = 0;
  mqttClient.publish((char*) stateTopic.c_str(), (char*) String(deviceState).c_str());
  Serial.println("device disabled.");
}

void callback(char* topic, uint8_t* payload, uint length)
{
  Serial.println("-------------------------------");
  Serial.print("New message arrived in topic: ");
  Serial.println(topic);

  for (uint i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
```

```
 }

 Serial.println();

 String strTopic = String(topic);

 if(strTopic == pingTopic) {
  Serial.println("Got ping ...");
  mqttClient.publish((char*) pongTopic.c_str(), (char*) deviceId.c_str());
  Serial.println("Sent pong ...");
 }

 if(strTopic == controlTopic) {
  int newState = (int)payload[0] - 48;
  Serial.print("Got control command: ");

  if(newState == 0) {
   disable_security();
  }
  else {
   if(newState == 1) {
    enable_security();
   }
   else {
    Serial.println("unknow command -> do nothing.");
   }
  }
 }
}

void setup_mqtt()
{
   mqttClient.setServer(mqtt_server, mqttPort);
   mqttClient.setCallback(callback);

   while (!mqttClient.connected()) {
      Serial.println("Connecting to MQTT...");

      if (mqttClient.connect("ESP32 Client")) {
         Serial.println("MQTT connected!");
      }
      else {
         Serial.print("MQTT connection failed with statu: ");
         Serial.println(mqttClient.state());
         delay(1000);
      }
   }

   mqttClient.subscribe((char*) pingTopic.c_str());
   mqttClient.subscribe((char*) controlTopic.c_str());
```

```
mqttClient.publish((char*) authTopic.c_str() , (char*) String(deviceType).c_str());
}

void setup_pins() {
  pinMode(pinDeviceState, OUTPUT);
  pinMode(pinAlarmState, OUTPUT);
  pinMode(pin1, INPUT_PULLUP);
  // End of start() set stat
  digitalWrite(pinDeviceState, HIGH);
}

void check_device_state() {
  if(deviceState == 0) {
    digitalWrite(pinAlarmState, LOW);
  }

  if(deviceState == 1) {
    digitalWrite(pinAlarmState, HIGH);
  }

  if(deviceState == 2) {
    if(digitalRead(pinAlarmState) == 1) {
      digitalWrite(pinAlarmState, LOW);
    }
    else {
      digitalWrite(pinAlarmState, HIGH);
    }
  }

  if(deviceState == 3) {
    Serial.println("-----------------------------");
    Serial.println("Detected sensor triggering.");
    deviceState = 2;
    Serial.println("Set device status: alarm.");
    detachInterrupt(pin1);
    Serial.println("Pin detached");
    mqttClient.publish((char*) alarmTopic.c_str(), (char*) String(deviceId).c_str());
    Serial.println("Sent alarm message.");
  }

  delay(500);
}

void setup() {
  setup_init();
  setup_wifi();
  setup_mqtt();
  setup_pins();
}
```

```
void loop() {
  mqttClient.loop();
  check_device_state();
}
```

ПРИЛОЖЕНИЕ Г
МОДУЛЬ УСТРОЙСТВА УПРАВЛЕНИЯ

Листинг Г.1 – файл «smart.js»

```
const express = require("express");
const {check, validationResult} = require('express-validator/check')
const router = express.Router();
const storage = require("node-persist");
const logger = require("../logger");
const stringGenerator = require("randomstring");
const amqplib = require('amqplib');

let devices = null;
let groups = null;
let pongRespinse = [];

/**
 * Storages initialisation
 */
const check_storage = async () => {
    if (!devices) {
        logger.info("Initialise devices storage.");
        devices = await storage.create({dir: "./storage/devices"});
        await devices.init();
    }

    if (!groups) {
        logger.info("Initialise groups storage.");
        groups = await storage.create({dir: "./storage/groups"});
        await groups.init();
    }
};

/**
 * Notification queue
 */
const pushTopic = "push";

_send_push = async (data) => {
    const open = require('amqplib').connect('amqp://localhost');
    open.then(function(conn) {
        return conn.createChannel();
    }).then(function(ch) {
        return ch.assertQueue(pushTopic).then(function(ok) {
            return ch.sendToQueue(pushTopic, Buffer.from(data);
        })
            .then(r => logger.info(`R: ${r}`));
```

```
  }).catch(console.warn);
};

/**
 * MQTT
 */
const mqtt = require('mqtt');
const mqttClient = mqtt.connect('mqtt://localhost');
const testTopic = "presence";
const pingTopic = "/ping";
const pongTopic = "/pong";
const authTopic = "/auth/#";
const stateTopic = "/state/#";
const alarmTopic = "/alarm";
const controlTopic = "/control/";
const authRegularStr = "^(\/auth\/)[0-9]{3}$";
const stateRegularStr = "^(\/state\/)[0-9]{3}$";

const _subscribeTopic = (topic) => {
    mqttClient.subscribe(topic, (err) => {
        if (!err) {
            logger.info(`Subscribed to ${topic} topic.`);
        } else {
            logger.error(`Subscription to "${topic}  topic failed.`);
        }
    });
};

mqttClient.on('connect', () => {
    mqttClient.subscribe(testTopic, (err) => {
        if (!err) {
            logger.info(`Connected to MQTT broker.`);
            check_ping_pong();
        } else {
            logger.error(`Error MQTT broker connection ...`);
        }
    });

    mqttClient.publish(testTopic, 'Test MQTT hello message ...');

    // Our subscriptions
    _subscribeTopic(authTopic);
    _subscribeTopic(pongTopic);
    _subscribeTopic(stateTopic);
    _subscribeTopic(alarmTopic);
});

const _handelAuth = async (id, type) => {
    let deviceInfo = await devices.getItem(id);
```

```
 if (!deviceInfo) {
        deviceInfo = {id: id, type: type};
        await devices.update(id, deviceInfo);
        logger.info(`New device ${id} with type: ${type}`);
        await _setState(id, "0");
    } else {
        logger.info(`Device: ${id} already exists`);
    }
};

const _setState = async (id, state) => {
    const deviceInfo = await devices.getItem(id);
    deviceInfo.state = state;
    await devices.update(id, deviceInfo);
    logger.info(`Set state: ${state} for: ${id}`);
};

mqttClient.on('message', async (topic, message) => {
    logger.debug(`New message: ${message} received in topic ${topic}`);
    await check_storage();

    if (topic === alarmTopic) {
        logger.warn(`Got alarm from: ${message}`);
        await _send_push(message);
    } else {
        if (topic === pongTopic) {
            const deviceId = message.toString('utf8');
            logger.info(`Pong: ${deviceId}`);
            pongRespinse.push(deviceId);
        } else {
            if (topic.match(stateRegularStr)) {
                const deviceId = topic.substr(7, 3);
                const deviceState = message.toString('utf8');
                logger.info(`Got new state ${deviceState} from ${deviceId}`);
                await _setState(deviceId, deviceState);
            } else {
                if (topic.match(authRegularStr)) {
                    const deviceId = topic.substr(6, 3);
                    const deviceType = await message.toString('utf8');
                    logger.info(`Got new authentication from: ${deviceId} with type ${deviceType}`);
                    await _handelAuth(deviceId, deviceType);
                    pongRespinse.push(deviceId);
                }
            }
        }
    }
});

/**
 * Check device (ping/pong)
```

```
 */
const check_ping_pong = async () => {
    await check_storage();
    pongRespinse = [];
    mqttClient.publish(pingTopic, "");
    logger.info(`Ping ...`);

    setTimeout(async () => {

        let deviceCount = 0;
        let connectedCount = 0;

        logger.debug(JSON.stringify(await devices.keys()));
        logger.debug(JSON.stringify(pongRespinse));

        await devices.forEach(async (device) => {
            // Check all connected devices
            logger.debug(`Check ${JSON.stringify(device)}`);

            if (pongRespinse.indexOf(device.key) == -1) {
                // detect disconnected device
                if (device.value.state !== "-1") {
                    logger.warn(`device ${device.key} disconnected`);
                    await _setState(device.key, "-1");
                    await _send_push(await devices.getItem(device.key));
                }
            } else {
                // detect connected device
                connectedCount++;

                if (device.value.state == "-1") {
                    logger.warn(`Device ${device.key} reconnected`);
                    await _setState(device.key, "0");
                    await _send_push(await devices.getItem(device.key));
                }
            }

            deviceCount++;
        });

        logger.info(`Device connection: ${connectedCount}/${deviceCount}`);

        await check_ping_pong();
    }, 10000);
};

const _wait = (ms) => {
    return new Promise(resolve => {
        setTimeout(resolve, ms);
    });
```

```
};

const _mqttSetDevicesState = async (group, state) => {

    logger.warn(JSON.stringify(group));

    for (const id of group.devices) {
        mqttClient.publish(controlTopic + id, state);
    };

    await _wait(1000);
    const response = [];

    logger.warn(JSON.stringify(group));
    for (const id of group.devices) {
        const device = await devices.getItem(id);

        logger.warn(JSON.stringify(device));
        if (device.state !== state) {
            logger.error(`Some device has not rent response`);
            await _send_push(device);
        };

        response.push(device);
    };

    return response;
};

const _mqtt_enable_devices = async (group) => {
    return await _mqttSetDevicesState(group, "1");
};

const _mqtt_disable_devices = async (group) => {
    return await _mqttSetDevicesState(group, "0");
};


// API
// 1 - Get groups
router.get('/group',
    async (req, res) => {
        await check_storage();
        return res.status(200).send({
            groups: await groups.values(),
            devices: await devices.values()
        });
    }
);
```

```
// 2 - Create group
router.post('/group', [
        check('name').isString().withMessage("Must be a string")
            .isLength({min: 10}).withMessage("Must be at least 10 characters long.")
    ],
    async (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(422).json({errors: errors.array()});
        }

        const newGroupId = stringGenerator.generate(10);
        await groups.setItem(newGroupId, {id: newGroupId, name: req.body.name, devices: []});
        logger.info(`Created new group ${newGroupId} with name: ${req.body.name}`);
        return res.status(201).send(newGroupId);
    });

// 3 - Remove group
router.delete('/group', [
        check('id').isString().withMessage("Must be a string")
            .isLength({min: 10, max: 10}).withMessage("Must be 10 characters long")
    ],
    async (req, res) => {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(422).json({errors: errors.array()});
        }

        logger.warn(req.body.id);
        const group = await groups.getItem(req.body.id);
        if (!group) {
            return res.status(404).json({errors: "Group not found"});
        }

        await groups.removeItem(req.body.id);
        logger.info(`Group ${req.body.id} removed`);
        return res.status(200).send();
    });

// 4 - ON group
router.post('/group/on', [
        check('id').isString().withMessage("Must be a string")
            .isLength({min: 10, max: 10}).withMessage("Must be 10 characters long")
    ],
    async (req, res) => {
        const errors = validationResult(req);

        if (!errors.isEmpty()) {
            return res.status(422).json({errors: errors.array()});
        }
```

```
    const group = await groups.getItem(req.body.id);

    if (!group) {
      return res.status(404).json({errors: "Group notfound"});
    }

    if (group.devices.length == 0) {
      return res.status(404).json({errors: "Group is empty"});
    }

    return res.status(200).json({
      id: group.id,
      name: group.name,
      devices: await _mqtt_enable_devices(group)
    });
  });

// 5 - OFF group
router.post('/group/off', [
    check('id').isString().withMessage("Must be a string")
      .isLength({min: 10, max: 10}).withMessage("Must be 10 characters long")
  ],
  async (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(422).json({errors: errors.array()});
    }

    const group = await groups.getItem(req.body.id);

    if (!group) {
      return res.status(404).json({errors: "Group not found"});
    }

    if (group.devices.length == 0) {
      return res.status(422).json({errors: "Group is empty"});
    }

    return res.status(200).json({
      id: group.id,
      name: group.name,
      devices: await _mqtt_disable_devices(group)
    });

  });

// 6 - Add device to group
router.post('/group/devices', [
```

```
    check('groupId').isString().withMessage("Must be a string")
      .isLength({min: 10, max: 10}).withMessage("Must be 10 characters long"),
    check('deviceId').isNumeric().withMessage("Must be a number")
      .isLength({min: 3, max: 3}).withMessage("Must be 3 digit number.")
  ],
  async (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(422).json({errors: errors.array()});
    }

    const groupId = req.body.groupId;
    const deviceId = req.body.deviceId;
    const group = await groups.getItem(groupId);

    if (!group) {
      return res.status(404).json({errors: "Group now found"});
    }

    const device = await devices.getItem(deviceId);

    if (!device) {
      return res.status(404).json({errors: "Device now found"});
    }

    if (group.devices.indexOf(deviceId) !== -1) {
      return res.status(422).json({errors: "Device already in group"});
    }

    group.devices.push(deviceId);
    await groups.update(groupId, group);

    return res.status(200).send();
  });

// 7 - Remove device from group
router.delete('/group/devices', [
    check('groupId').isString().withMessage("Must be a string")
      .isLength({min: 10, max: 10}).withMessage("Must be 10 characters long"),
    check('deviceId').isNumeric().withMessage("Must be a number")
      .isLength({min: 3, max: 3}).withMessage("Must be 3 digit number.")
  ],
  async (req, res) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      return res.status(422).json({errors: errors.array()});
    }
```

```
    const groupId = req.body.groupId;
    const deviceId = req.body.deviceId;
    const group = await groups.getItem(groupId);

    if (!group) {
      return res.status(404).json({errors: "Group now found"});
    }

    const device = await devices.getItem(deviceId);

    if (!device) {
      return res.status(404).json({errors: "Device now found"});
    }

    const deviceIndex = group.devices.indexOf(deviceId);

    if (deviceIndex == -1) {
      return res.status(404).json({errors: "Device not in the group"});
    }

    await group.devices.splice(deviceIndex, 1);
    logger.warn(JSON.stringify(group.devices));
    await groups.update(groupId, group);

    return res.status(200).send();
  });

module.exports = router;
```

ПРИЛОЖЕНИЕ Д
ДОСТАВКА PUSH УВЕДОМЛЕНИЙ

Листинг Д.1 – файл «push.py»

```
import json
import logging
import pika
import requests

from configs.config import CONFIG

connection =
pika.BlockingConnection(pika.ConnectionParameters(host=CONFIG["rabbitmq"]["host"]["internal"]))
channel = connection.channel()
channel.queue_declare(queue=CONFIG["rabbitmq"]["queue"]["push"], durable=True)

def callback(ch, method, properties, body):
    req = requests.session()
    req.headers.clear()
    headers = {
        'Authorization': CONFIG["firebase"]['messaging']['key'],
        'Content-Type': 'application/json'
    }
    req.headers.update(headers)
    url = "https://fcm.googleapis.com/fcm/send"

    body_json = json.loads(body)
    request_body = {
        "to": body_json["tokens"],
        "collapse_key": 'type_a',
        "data": {
            "title": body_json['n_title'],
            "desc": body_json['n_desk'],
            "image_name": body_json['n_image_name'],
            "image_url": body_json['image_url']
        }
    }

    resp = req.post(url=url, json=request_body)
    logging.info(resp)
    ch.basic_ack(delivery_tag=method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue=CONFIG["rabbitmq"]['queue']["push"])

print('Waiting for messages...')
channel.start_consuming()
```

ПРИЛОЖЕНИЕ Е
РАССЫЛКА EMAIL

Листинг Е.1 – файл «email.py»

```
import json
import pika
import smtplib

from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email.mime.image import MIMEImage
from configs.config import CONFIG

connection =
pika.BlockingConnection(pika.ConnectionParameters(host=CONFIG["rabbitmq"]["host"]["internal"]))
# connection = pika.BlockingConnection(pika.ConnectionParameters(host="localhost"))
channel = connection.channel()

channel.queue_declare(queue=CONFIG["rabbitmq"]['queue']["email"], durable=True)

msg = MIMEMultipart()

password = CONFIG['email']['password']
msg['From'] = CONFIG['email']['login']
msg['Subject'] = "Security alarm"

server = smtplib.SMTP('smtp.gmail.com: 587')
server.starttls()
server.login(msg['From'], password)


def callback(ch, method, properties, body):
    image_path = "human2.jpg"
    body_json = json.loads(body)

    from helpers.images.ImageProcessor import str_to_image
    str_to_image(body_json['image'], image_path)

    print(body_json["to"][0])
    msg['To'] = body_json["to"]
    msg.attach(MIMEText("New message", 'plain'))

    with open(image_path, 'rb') as image:
        msg.attach(MIMEImage(image.read()))

    server.sendmail(msg['From'], msg['To'], msg.as_string())
```

```
 print("successfully sent email to %s:" % (msg['To']))
    ch.basic_ack(delivery_tag=method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue=CONFIG["rabbitmq"]['queue']["email"])
channel.start_consuming()
```

ПРИЛОЖЕНИЕ Ж
ОТПРАВКА УВЕДОМЛЕНИЙ В TELEGRAM

Листинг Ж.1 – файл «telegram.py»

```
import telebot
from telebot import apihelper
import json
import pika
from configs.config import CONFIG

print(CONFIG["telegram"]["proxy"])
print(CONFIG["telegram"]["bot"]["token"])

apihelper.proxy = {'https': CONFIG["telegram"]["proxy"]}
my_telegram_bot = telebot.TeleBot(CONFIG["telegram"]["bot"]["token"])


from docker.telegram.bot.TelegramBot import my_telegram_bot
from helpers.images.ImageProcessor import str_to_image

connection =
pika.BlockingConnection(pika.ConnectionParameters(host=CONFIG["rabbitmq"]["host"]["internal"]))
# connection = pika.BlockingConnection(pika.ConnectionParameters(host="localhost"))
channel = connection.channel()

channel.queue_declare(queue=CONFIG["rabbitmq"]["queue"]["telegram"], durable=True)

def callback(ch, method, properties, body):
    image_path = "human1.jpg"
    body_json = json.loads(body)

    str_to_image(body_json['image'], image_path)

    my_telegram_bot.send_message(body_json['to'], body_json['message'])

    with open(image_path, 'rb') as image:
        my_telegram_bot.send_photo(body_json['to'], image)

    ch.basic_ack(delivery_tag=method.delivery_tag)


channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue=CONFIG["rabbitmq"]["queue"]["telegram"])
channel.start_consuming()
```