МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«КРЫМСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ имени В. И. ВЕРНАДСКОГО»
Физико-технический институт
Кафедра информатики

Гончаров Артём Максимович

ФУНДАМЕНТАЛЬНАЯ АРХИТЕКТУРА И СТРУКТУРА ПРОТОКОЛА СВЯЗИ ОБЛАЧНОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ ТИПА «КЛИЕНТ-НА-СЕРВЕРЕ»

Научная работа

обучающегося 1 курса

направления подготовки 01.04.02. Прикладная математика и информатика

форма обучения очная

Научный руководитель
доцент кафедры информатики,
кандидат технических наук А. И. Козлов

Симферополь, 2024

Работа посвящена нахождению и практической реализации методов осуществления глубокой синхронизации клиентских пользовательских устройств. Такая синхронизация осуществляется при помощи облачной операционной системы типа «клиент-на-сервере». В ходе работы была выработана фундаментальная архитектура системы и разработаны способы взаимодействия между её компонентами. Для обеспечения однородной внутрисистемной коммуникации очень широкого спектра пользовательских устройств с сервером разработан универсальный протокол связи.

Ключевые слова: облачная ОС, клиент-на-сервере, глубокая синхронизация, протокол связи.

Страниц – 55, таблиц – 0, иллюстраций – 7, приложений – 6, библиографических источников – 19.

Оглавление

Введение

Современные операционные системы позволяют успешно и эффективно использовать аппаратные возможности устройств. Однако, они не обеспечивают связности и скоординированности действий между индивидуальными пользовательскими устройствами. Для пользователя является естественным иметь несколько наборов устройств разных типов, таких как телефоны, компьютеры, смарт-часы, телевизоры и т.д. Отсутствие поддержки глубокой синхронизации их операционными системами приводит к привязанности к конкретному устройству и невозможностью беспрепятственного продолжения рабочего процесса пользователя после перехода с одного устройства на другое.

В течение первого десятилетия 21-го века происходило стремительное и скачкообразное развитие мобильных и стационарных пользовательских устройств [1]. Производители устройств регулярно вносили революционные изменения в то, как пользователь взаимодействует со своим устройством. За очень короткий срок это обеспечило трансформацию первых переносных кнопочных телефонов в многофункциональные смартфоны под управлением операционных систем Android и IOS, телевизоров, предназначенных только для демонстрации изображения из эфира, в Smart-телевизоры, первых планшетных компьютеров под управлением Microsoft Tablet PC в современные интернет-планшеты.

В годы, последовавшие за этим периодом активного развития, устройства претерпевали лишь незначительные с точки зрения пользователя аппаратные и дизайнерские изменения. Значимые изменения в сторонних сервисах и приложениях, используемых на пользовательских устройствах, возникали не под влиянием новых интерфейсов взаимодействия, а опосредованно. Большинство существовавших на конец описанного периода операционных систем также не изменились существенно до наших дней.

В то время, как развитие пользовательских устройств общего назначения (компьютер, смартфон, и т.д.) резко замедлило свой темп, устройства интернета вещей (англ. internet of things, IoT) [2] и умного дома стали многочисленными и практически повсеместными. Однако, на сегодняшний день не существует единого способа управления всеми этими устройствами и объединяющего их протокола связи. Ситуация усложняется тем, что контроль над устройствами IoT и умного дома должен эффективно осуществляться в том числе устройствами общего назначения,

следовательно, все эти три группы устройств должны быть объединены в одну систему.

Для полноценного разрешения всех вышеописанных проблем необходим полный пересмотр концепции, архитектуры и способов построения пользовательских операционных систем и разработка их новых типов. Создание новых технологий взаимодействия пользователя со своими устройствами обеспечит изменение как программных, так и аппаратных составляющих самих устройств и послужит толчком для нового периода «взрывного» развития ориентированных на удобство пользователя компьютерных технологий. Этим обеспечивается актуальность работы, которая рассматривает аспекты построения операционной системы типа «клиент-на-сервере».

Целью работы является определение концептуальных компонентов и проектирование фундаментальной архитектуры облачной операционной системы типа «клиент-на-сервере» и разработка механизма передачи данных пользовательских устройств при помощи протокола связи, который лежит в основе такой операционной системы. Для достижения этой цели были поставлены следующие задачи:

- проанализировать современные операционные системы, выделить их преимущества и недостатки;
- исследовать современные возможности реализации неглубокой синхронизации;
- изучить спектр возможностей, предоставляемых облачным размещением операционной системы;
- определить понятие типа операционной системы «клиент-на-сервере»;
- установить требования к операционным системам такого типа;
- разработать концепцию компонентов фундаментальной архитектуры операционной системы;
- выделить подходы и методы её реализации;
- описать способы и принципы коммуникации устройств операционной системы по протоколу связи;
- разработать механизмы для передачи данных физических пользовательских устройств на сервер при помощи протокола связи операционной системы;
- имплементировать некоторые базовые компоненты протокола соединения.

1. Перспективы развития операционных систем

1.1. Операционные системы с точки зрения пользователя, их преимущества и недостатки

Согласно [3] любое программируемое устройство, не запрограммированное на аппаратном уровне на выполнение конкретной задачи, должно иметь операционную систему – программную прослойку между аппаратным компонентом (hardware) и программным компонентом (software). Попробуем расширить это определение операционных систем, исследуя их с точки зрения пользователя. Рассмотрим взаимодействие обычного пользователя с набором программируемых устройств. Зачастую он неглубоко осведомлён о концептуальной архитектуре устройства, с которым взаимодействует. В некоторых случаях, например, при работе с бытовой техникой, пользователь может вообще не определить наличие операционной системы. В случае же взаимодействия с программируемым устройством общего назначения (компьютер, смартфон, и т.д.) наличие операционной системы становится очевидно для пользователя. Это высказывание на самом деле справедливо только для тех компонентов ОС, с которыми пользователь непосредственно взаимодействует. К таким компонентам относятся графический интерфейс, файловая система, система запуска приложений и другие, однако, к ним не относятся, например, системы управления процессорным временем и виртуальной оперативной памятью [4].

Учитывая вышесказанное, можно сформировать следующее определение операционной системы, основанное на пользовательской точке зрения: операционная система – это комплексный модульный механизм, позволяющий высокоуровневое взаимодействие набора ориентированных на пользователя приложений со структурой, проводящей низкоуровневые вычисления. Как можно видеть, такой вариант определения не вступает в конфликт с устоявшимся и широко принятым. Однако, учитывая специфику взаимодействия пользователей с устройствами, определение позволяет сместить акцент рассмотрения структуры ОС с аппаратно-ориентированных деталей на общие черты пользовательского опыта (user experience). Основываясь на данном определении, рассмотрим преимущества и недостатки современных операционных систем общего назначения.

К преимуществам современных ОС общего назначения относятся:

1) удобный интерфейс: операционные системы общего назначения предоставляют разнообразные графические и текстовые интерфейсы,

которые делают взаимодействие пользователя с устройством удобным и интуитивно понятным. Дополнительные функции, такие как многозадачность, возможность множественных окон и жестов управления, делают работу с операционной системой более эффективной и продуктивной;

2) множество приложений: современные операционные системы поддерживают большое количество приложений различных типов и назначений. Это позволяет пользователям выбирать и устанавливать программы, соответствующие их потребностям, такие как офисные приложения, браузеры, мультимедийные проигрыватели, игры и многое другое. Возможность дополнительной установки приложений делает операционные системы более универсальными и адаптированными под различные сферы деятельности;

3) поддержка аппаратного обеспечения: современные ОС общего назначения предоставляют драйверы для большинства распространенных аппаратных устройств, таких как принтеры, сканеры, видеокарты, микрофоны и другие. Это обеспечивает совместимость с широким спектром аппаратного обеспечения и позволяет пользователям использовать различные устройства с операционной системой без необходимости установки дополнительных драйверов;

4) обновления и поддержка: операционные системы регулярно обновляются разработчиками, что позволяет исправлять ошибки, улучшать функциональность и обеспечивать безопасность системы. Кроме того, операционные системы обычно имеют широкую поддержку со стороны разработчиков, включая официальные форумы поддержки, документацию и информационные уведомления.

К недостаткам современных ОС общего назначения относятся:

1) безопасность: современные операционные системы общего назначения подвержены угрозам безопасности, таким как вирусы, вредоносное ПО, хакерские атаки и т. д. Несмотря на меры безопасности, встроенные в операционные системы, некорректная конфигурация или неправильное использование системы может привести к уязвимостям и нарушению безопасности данных и личной информации пользователей;

2) отсутствие глубокой синхронизации: на протяжении времени развития операционных систем общего назначения были предприняты успешные попытки синхронизировать отдельные категории данных между различными пользовательскими устройствами. Далее, ряд таких

механизмов синхронизации был интегрирован в структуру некоторых современных операционных систем (Chrome OS) [5]. Однако, состояние общей и универсальной синхронизации между пользовательскими устройствами так и не было достигнуто;

3) аппаратная зависимость: в зависимости от аппаратной платформы и класса материнского устройства варьируются доступные функции, характер функционирования и концептуальная структура операционных систем. Таким образом, используя разные по структуре классы устройств пользователь получает в корне разный интерфейс взаимодействия. Также пользователь вынужден выбирать необходимое устройство среди других устройств, учитывая ширину функционала его операционной системы;

4) системные требования: операционная система и приложения, функционирующие на её основе, предоставляют требования к аппаратному обеспечению. Выполнение широкого набора таких требований, в свою очередь, ведёт к удорожанию устройства и одновременному усложнению его аппаратного компонента;

Выделение преимуществ и недостатков современных широко используемых операционных систем является необходимым промежуточным шагом для нахождения методов исправления недостатков и расширение круга преимуществ. Одним из возможных решений является разработка структуры операционной системы нового поколения.

1.2. Облачное размещение и концепция «клиент-на-сервере»

Рассмотрим набор взаимно независимых решений, устраняющий ряд недостатков современных операционных систем общего назначения. Так, при помощи технологии виртуальных машин операционная система может быть установлена в изолированный виртуальный контейнер. Такой контейнер позволяет виртуализировать аппаратные компоненты компьютера-носителя. Таким образом, может быть полностью решена проблема аппаратной зависимости современных операционных систем. Также, использование виртуальных машин позитивно влияет на безопасность виртуализируемых операционных систем [6].

Сервисы синхронизации файлов, такие как Microsoft OneDrive, Yandex.Disc и другие подобные сервисы, могут быть использованы как основа для системы, интегрирующей технологию глубокой синхронизации. Это суждение верно лишь в том случае, если за основную структуру хранения данных в операционной системе принимать файловую систему.

Отдельного внимания в контексте исправления недостатков современных операционных систем общего назначения заслуживают сервисы и протоколы удалённого доступа к устройствам. К примерам можно отнести RDP, SSH, AnyDesk, TeamViewer, и другие [7, 8]. Будем считать исследуемой операционную систему устройства, удалённый доступ к которому обеспечивается, а не операционную систему подключающегося устройства. Тогда, при использовании удалённого подключения для операционной системы выполняется глубокая синхронизация. При инициации удалённого подключения с любого стороннего устройства весь комплекс пользовательских данных и общая функциональность операционной системы остаются неизменными. В качестве компенсации, если протокол удалённого подключения поддерживает графический интерфейс, может происходить потеря его удобства и адаптивности для пользователя.

Заметим, что современное применение инструментов синхронизации определённых компонентов операционных систем не подразумевает коренного изменения структуры самих ОС. Синхронизация данных обеспечивается сторонними и узкоспециализированными приложениями. В подавляющем большинстве случаев между синхронизируемыми устройствами нет никакого порядка приоритета. Иными словами, набор приложений равноправных устройств обращается к общему (обычно удалённому) хранилищу данных для независимой обработки и последующего предоставления этих данных при помощи эквивалентного интерфейса. В остальных случаях, касающихся в основном беспроводных периферийных устройств, отдельные устройства пользователя начинают выполнять роль узлов синхронизации.

Значительно отличающаяся модель взаимодействия устройств была избрана при применении одной из моделей концепции облачных вычислений – SaaS (Software as a Service) [9]. При имплементации такой модели устройства-клиенты подключаются к размещённому на сервере сервису, который обеспечивает не только хранение, но и обработку данных. Таким образом, серверу предоставляется определённый авторитет, однако для подключения к SaaS-сервису на устройстве должна присутствовать полноценная операционная система, подверженная всё тем же недостаткам.

Проанализировав оба подхода к организации механизмов взаимодействия операционных систем устройств и синхронизации различных данных и рассмотрев различные частичные решения,

позволяющие устранить ряд недостатков современных ОС, можно прийти к выводу о необходимости облачного размещения операционных систем. Выделим те компоненты операционной системы общего назначения, которые неизбежно должны быть расположены локально для обеспечения взаимодействия между операционной системой и пользователем. К таким компонентам относятся системы ввода и вывода, а именно системы взаимодействия с устройствами ввода-вывода и первичной обработки данных, связанных с этими устройствами. Сохраняя только эти функции на множественных пользовательских устройствах, можно перенести операционную систему на сервер. Такое инженерное решение добавит к списку функций пользовательских устройств инициацию подключения к серверу и поддержание канала связи согласно установленному протоколу. Описанное распределение функций и структур не вполне соответствует клиент-серверной модели, так как клиентская сторона сервиса не выполняет сколько-нибудь значимых логических вычислений, а является структурой поддержания соединения. По этой причине и для выделения конкретного вышеописанного подхода к построению операционных систем от общего понятия облачной операционной системы, будем называть этот подход «клиент-на-сервере». Это означает, что большая часть традиционных функций устройства-клиента перенесена на сервер.

Применение фундаментальной архитектуры операционной системы «клиент-на-сервере» устраняет большинство недостатков современных операционных систем и расширяет их уже имеющиеся преимущества. Этот подход в полной мере реализует глубокую синхронизацию пользовательских устройств, устраняет аппаратную зависимость клиентских устройств и исключает большинство их системных требований. Приложения такой операционной системы могут быть адаптированы для любого пользовательского устройства с точностью до требуемых устройств ввода-вывода.

Заметим, что при использовании архитектуры «клиент-на-сервере» критерий, определяющий клиентское устройство, так широк, что нет необходимости ограничиваться устройствами общего назначения. Такие узкоспециализированные устройства как бытовая техника, периферийные компьютерные гаджеты и нательные устройства могут быть подключены напрямую к серверной части операционной системы, не используя другие клиентские устройства как посредников.

Установим список требований, которым должна удовлетворять ОС архитектуры «клиент-на-сервере»:

1) ОС должна поддерживать глубокую синхронизацию и обеспечивать взаимодействие устройств по определению своей архитектуры;
2) ОС должна обеспечивать одинаковый пользовательский опыт и функциональные возможности в пределах одного класса устройств;
3) ОС должна гарантировать полную конфиденциальность пользовательских данных и обеспечивать безопасность системы в целом;
4) ОС должна поддерживать максимально широкий спектр устройств, которые имеют возможность подключения к Интернету, и быть универсальной в отношении интерфейса взаимодействия с ними;
5) ОС должна позволить значительно снизить цены на пользовательские устройства;
6) ОС должна сделать пользовательские устройства более энергоэффективными, чтобы увеличить ёмкость и срок службы батареи устройства и снизить энергопотребление сети в целом;
7) ОС должна централизовать вычислительную мощность, чтобы уменьшить потребность в обслуживании на стороне пользователя, и предлагать широкий спектр аппаратного обеспечения по требованию пользователя;
8) ОС должна помочь в предотвращении изменения климата путем перевода централизованных вычислительных мощностей на возобновляемые источники энергии;
9) ОС должна предоставлять эффективный механизм монетизации.

Следует заметить, что на сегодняшний день практически все окружающие пользователя программируемые устройства имеют возможность подключения к интернету, а скорость домашнего и 5G сотового интернета достигает показателя в 1 Гбит/с [10]. Следовательно, ранее существовавшее ограничение качества интернет-соединения в виде задержек и низких скоростей не актуально. Это делает возможным повседневное пользование операционной системы архитектуры «клиент-на-сервере».

2. Фундаментальная архитектура ОС типа «клиент-на-сервере»

2.1. Концептуальные объекты фундаментальной архитектуры

Для реализации экземпляра операционной системы общего назначения типа «клиент-на-сервере», удовлетворяющего требованиям, установленным для систем такого типа, необходимо выделить его наиболее значимые индивидуальные компоненты. Также, необходимо ввести ряд определений.

Терминал (Terminal) является индивидуальным устройством пользователя и клиентом по отношению к серверу. Он имеет доступ к серверу через специализированный канал связи. Терминал инициирует соединение между собой и сервером. Основной задачей терминала является представление действий пользователя путем обслуживания устройств ввода-вывода и взаимодействия с ними. терминал в основном преобразует данные, полученные из реального мира, в логические входные данные и подготавливает их для отправки на сервер или, наоборот, представляет полученные от сервера логические выходные данные пользователю. Предполагается, что пользователь имеет несколько терминалов, функционирующих одновременно.

Сервер (Server) представляет собой удаленную единицу операционной системы. Каждый пользователь имеет не более одного сервера. Сервер воплощает в себе основные части и функции операционной системы, такие как управление процессами, памятью, файловой системой и временем выполнения приложений. К серверу одновременно подключено несколько пользовательских терминалов. Сервер запускает пространства, приложения и представителей ввода-вывода. Сервер поддерживает соединения между взаимно-однозначно сопоставленными представителями ввода-вывода и терминалами.

Представитель ввода-вывода (Input/Output Representative, I/O Rep.) представляет собой экземпляр терминала внутри сервера. Двусторонняя связь между терминалом и его представителем ввода/вывода имеет взаимно-однозначную природу и фундаментальное значение. Если в объяснении структуры сервера присутствует понятие представителя ввода-вывода, то оно в равной степени означает терминал на другой стороне канала связи и наоборот. Основная цель представителя ввода-вывода — получать и обрабатывать входные данные от терминала, действуя от его имени, а также получать данные вывода из приложений и ОС и отправлять их обратно

терминалу. Представитель ввода-вывода привязывается к пространству при подключении терминала к серверу и может перемещаться между пространствами, переопределяя первоначальную привязку. Тем не менее, представитель ввода-вывода является независимым объектом, который выполняет свои задачи автономно по отношению к пространству.

Пространство (Space) является единицей внутреннего взаимодействия сервера. Пространство представляет собой своеобразную объединяющую оболочку, связывающую между собой набор представителей ввода/вывода и приложений операционной системы. Для того, чтобы детально описать назначение пространства как компонента операционной системы необходимо рассмотреть общий характер взаимодействия между пользователем и приложениями операционной системы.

Операционная система типа «клиент-на-сервере» не предполагает одновременного использования несколькими физическими пользователями, зашедшими в систему под одним и тем же аккаунтом. Такое использование можно считать некорректным, так как из-за универсальности терминалов пользователи могут осуществлять быструю и полноценную смену пользовательского аккаунта путём инициации подключения к своему серверу.

Задача синхронизации процессов, вносящих изменения в структуру данных, к которой они имеют общий доступ записи, является совершенно нетривиальной задачей [11]. Её решение на практике требует значительных затрат. В случае, если данная задача всё же полностью решена, обновление используемых данных после уведомления о внесённых другим процессом изменений значительно усложняет структуру приложений операционной системы или может быть вообще невозможно. Таким образом, учитывая вышесказанное, в системе типа «клиент-на-сервере» следует запретить одновременное функционирование нескольких экземпляров одного и того же приложения, обрабатывающих идентичные данные. Например, допустимо одновременное функционирование процессов текстового редактора, взаимодействующего с разными документами, но недопустимо для тех же процессов, работающих с одним и тем же файлом. Разрешение одновременного функционирования нескольких экземпляров одного приложения, работающих с общими данными, бессмысленно и значительно усложняет фундаментальную архитектуру операционной системы.

Тем не менее, введение такого запрета без некоторых оговорок сужает возможности облачной операционной системы и является некоторым

следованием концепции современных операционных систем. Так как вне зависимости от того, какой терминал использует пользователь, приложение и все его данные расположены на сервере, при попытке повторно открыть ранее открытое приложение с тем же набором обрабатываемых данных пользователю может быть предложено переместить сеанс взаимодействия с приложением на актуальный терминал с ранее используемого терминала. Более того, основываясь на современной практике командного взаимодействия с операционной системой путём удалённого подключения (см. 2.2), можно отождествить определённые потоки ввода и вывода нескольких терминалов для взаимодействия с одним приложением. Таким образом, изменения, внесённые с одного терминала, будут немедленно отображены на другом, а со стороны приложения несколько терминалов будут неотличимы от одного.

Возвращаясь к вопросу определения концепции пространства, его непосредственной задачей является организация соответствующей группировки ввода-вывода приложений и представителей ввода-вывода. А именно, осуществление отождествления ввода-вывода терминалов, взаимодействия между ними, обеспечение подачи общих данных приложению и приём совмещённых данных от него. Также, пространство должно обеспечивать построение интерфейса, потому что приложение возвращает данные в универсальном формате для всех терминалов пространства, и необходимо адаптировать эти данные для каждого из терминалов. Это наиболее актуально для графического интерфейса, так как формы, размеры и методы ввода экранов терминалов могут качественно различаться. Задача адаптации интерфейсов не может быть полностью возложена на представителей ввода-вывода. Представитель ввода-вывода концептуально выполняет роль электронного двойника терминала и не решает общих задач операционной системы.

Введённые выше понятия концептуальных объектов облачной операционной системы типа «клиент-на-сервере» являются фундаментальным отличием этого типа операционной системы от современных локально-размещённых систем общего назначения. Эти объекты порождаются необходимостью пересмотра взаимодействия пользователя с операционной системой и их взаимоотношения определяют основные черты системы ввода-вывода.

2.2. Подход к имплементации операционной системы

Как уже было неоднократно указано ранее, облачная операционная система типа «клиент-на-сервере» не является операционной системой в традиционном понимании этого понятия. Такая операционная система фокусируется на обеспечение эффективного и удобного пользовательского опыта, и пренебрегает низкоуровневыми аппаратно-ориентированными деталями. Раскроем суть этого суждения. Расположенная на удалённом сервере серверная (основная часть) операционной системы не имеет задачи функционировать на основе принадлежащего пользователю аппаратного компонента.

Наиболее эффективным способом размещения этого сегмента операционной системы является частично изолированный контейнер, расположенный на одном из глобального набора связанных в сеть облачных вычислений серверов. Этот контейнер, согласно концепции контейнеризации, будет использовать общую для всех контейнеров аппаратную платформу. Также, контейнер может быть перемещён с одного сервера на другой без потери функциональности и времени, затраченного на простой. Такая операция может проводиться для балансировки нагрузки на сервис облачных вычислений или поддержания работоспособности отдельных контейнеров при поломке или плановом отключении одного из серверов сети.

С учётом этих технических деталей можно обнаружить, что серверный компонент операционной системы может быть реализован как обширное приложение, производящее двухстороннюю коммуникацию с терминалами. Применяя соответствующий подход ликвидируется необходимость имплементации низкоуровневых компонентов операционной системы вплоть до уровня коммуникаций между процессами.

Структура клиентской части операционной системы типа «клиент-на-сервере» – операционной системы терминала – является более соответствующей современному пониманию операционной системы. Для уменьшения объёма работ по имплементации ОС терминала предлагается использовать специально модифицированное ядро Linux. Заметим, что основной задачей терминала является поддержание соединения. Сжатие передаваемой информации в случае аппаратной поддержки этой операции может приводить к меньшим вычислительным затратам, чем передача информации без сжатия. Требования безопасности соединения обеспечивают необходимость шифрования любой передаваемой по

соединению информации. Таким образом, для достижения наиболее эффективного функционирования терминала необходимо наличие в его физической компоненте вычислительных модулей с аппаратной поддержкой шифрования и различных алгоритмов сжатия и декомпрессии данных. Это, в свою очередь, позволяет снизить нагрузку на основной процессор. Учитывая принципы фундаментальной архитектуры операционной системы в целом, можно сделать вывод, что системные требования для устройства терминала крайне низки. Это позволяет разработчикам устройств сосредоточится на дизайне и удобстве пользователя, при этом одновременно кардинально снижая цену устройства.

Наиболее рациональным решением является реализация операционной системы типа «клиент-на-сервере» в объектно-ориентированном стиле. Концепция операционной системы в целом представляет собой набор слабо зависимых взаимодействующих объектов. Объектно-ориентированный подход к программированию не только соответствует этой концепции, но и является наиболее эффективным для разработки достаточно сложного программного обеспечения. Процедуры операционной системы вызываются с большой частотой и в больших объёмах. Поэтому для имплементации операционной системы не подходят интерпретируемые языки программирования. Для реализации ОС как основной нами был выбран язык C++. Он удовлетворяет всем вышеописанным требованиям.

Основной структурой хранения пользовательских и системных данных в современных операционных системах является файловая система. Её основными недостатками являются низкая скорость операции поиска и отсутствие встроенной и повсеместно используемой в системе структуризации данных. Меняя концептуальный подход к операционным системам в целом, стоит изменить подход к структуре хранения данных. Реляционная база данных устраняет вышеуказанные недостатки. К тому же, при условии облачного размещения ОС, она позволяет хранить данные большого числа пользователей более эффективно, чем множество изолированных файловых систем. Используя реляционную базу данных, можно легко эмулировать файловую систему. Это достигается следующим образом: каждый файл рассматривается как запись в базе данных, имеющая столбец, который указывает путь. Следовательно, изменяя подход к структуре хранения данных, мы обеспечиваем расширение возможностей при минимизации затрат и проводим некую стандартизацию.

3. Протокол соединения клиентских терминалов с сервером

3.1. Используемые технологии и общая структура протокола соединения

Центральным компонентом операционной системы типа «клиент-на-сервере» является протокол соединения между терминалами и сервером, который обеспечивает передачу данных ввода-вывода между структурными частями ОС. Такой протокол должен предусматривать механизм передачи данных ввода от момента их генерации устройством ввода до момента получения приложением в виде определённой структуры. Также должен быть предусмотрен механизм передачи и обработки данных вывода от момента их генерации операционной системой или приложением до момента представления пользователю в виде каких-то физических явлений. Для удовлетворения базовых требований к облачной операционной системе типа «клиент-на-сервере» необходимо, чтобы протокол связи был универсальным относительно любых типов клиентских устройств и видов передаваемых данных. Будем называть разрабатываемый нами протокол PODCAST (Portable On-Demand Client Access Service Technology, Технология сервиса портативного доступа клиента по востребованию).

Структурообразующей частью протокола соединения должен служить достаточно низкоуровневый способ передачи данных по интернету. Для этой цели хорошо подходит интерфейс сокетов Беркли. В практических целях было решено использовать библиотеку Windows Sockets 2, адаптированную под языки программирования C и C++. Для эффективной и удобной передачи данных соединения необходимо создать класс-обёртку для библиотеки. Это позволит при необходимости сменить библиотеку взаимодействия с сокетами, не изменяя внутренних интерфейсов операционной системы. Таким образом также достигается соответствие объектно-ориентированному стилю программирования операционной системы, производится стандартизация интерфейса коммуникации с помощью сокетов и расширяется функционал библиотеки.

Способ построения операционной системы, при котором все устройства пользователя, даже малогабаритные датчики, должны иметь переносимый и портативный способ подключения к серверу по сети Интернет, явно подразумевает использование исключительно IPv6 и полное пренебрежение понятием локальной сети (все устройства и так связаны в

сеть сервером). Однако, на период разработки тестовой и демонстрационной ограниченной версии протокола, условимся применять IPv4.

Для обеспечения требования к операционной системе о безопасности и приватности пользовательских данных, любые данные, передаваемые по каналу соединения, должны быть зашифрованы. Нельзя однозначно выделить подмножество таких данных ввода-вывода, которые для злоумышленника невозможно использовать любым способом. Для шифрования соединения был выбран алгоритм симметричного шифрования AES, который является стандартом эффективности, скорости и надёжности [12]. Было принято решение использовать криптографическую библиотеку Crypto++ для имплементации алгоритмов шифрования в ОС.

Однако, для первоначальной безопасной передачи симметричного ключа шифрования требуется алгоритм ассиметричного шифрования. Одним из популярных и надёжных алгоритмов такого типа является алгоритм RSA [13]. Введём термины клиента RSA, который имеет только открытый ключ, и сервера RSA, который имеет как открытый, так и закрытый ключ. Реализуем абстрактные классы для клиентских и серверных сущностей алгоритмов AES и RSA, руководствуясь теми же принципами, что и при создании обёртки библиотеки сокетов.

Для каждого соединения на сервере и терминале создаётся по одному объекту класса AES. Один из них генерирует ключ, а второй получает его. Стоит заметить, что для повышения безопасности имеет смысл генерировать ключ AES-шифрования каждый раз при установлении соединения. Напротив, ключи RSA-шифрования могут не изменяться в течение полгода-года. Поэтому, функция генерации ключей AES-шифрования всегда выполняется на сервере, более мощном, чем терминал. Библиотека для алгоритма RSA включает в себя возможность передачи информации в обе стороны для того, чтобы позволить терминалу запрашивать регенерацию ключей у сервера.

В комбинации механизм сокетов, по которому передаётся зашифрованная алгоритмом AES информация, и алгоритм RSA, использующийся для передачи ключей симметричного шифрования, позволяют создать эффективный и безопасный базис канала связи, являющегося одним из центральных элементов операционной системы типа «клиент на сервере».

3.2. Установка моста соединения

Этот раздел описывает как устанавливается соединение между Терминалом и Сервером, когда они уже сопряжены. Упомянутое сопряжение означает, что стороны предварительно настроены для совместного использования. Будем называть соединение, используемое для передачи неструктурированных потоковых данных мостом соединения.

Обычно существует два состояния, в которых может находиться терминал: изолированный режим и подключенный режим. Изолированный режим подразумевает состояние, в котором терминал заведомо не будет отправлять какие-либо данные на сервер и будет игнорировать все входящие данные, даже если они были отправлены. Это состояние соответствует выключенному или находящемуся в глубоком сне терминалу. Подключённый режим обозначает состояние, когда разрешен весь или частичный трафик данных. Активный режим и неглубокий сон указывают на подключенный режим. Неглубокий сон — это состояние, при котором терминал не используется в полной мере, но может быть разбужен при возникновении события. Например, у смартфона подсвечивается экран и издается звук при обнаружении входящего звонка или уведомления.

Чтобы установить мост соединения, когда терминал переходит в подключенное состояние, он инициирует соединение сокета. Сокет должен быть потоковым. Это означает, что он работает по протоколу TCP, гарантируя тем самым согласованность и структурную целостность данных, передаваемых по мосту. Сокет поддерживается в течение промежутка, который заканчивается, когда пользователь или сервер подает команду терминалу перейти в состояние изолированного режима. Если соединение было каким-то образом прервано, то предполагается, что терминал перешел в изолированный режим с последующим выходом из него, после чего соединение инициируется заново. Сокет следует защитить от спам-атак [14].

Когда мост готов к работе, терминал отправляет свой открытый ключ асимметричного шифрования (RSA), а сервер создает, шифрует и отправляет обратно ключ симметричного шифрования (AES), который дешифруется терминалом. С этого момента все данные, передаваемые между сторонами, надежно шифруются и обеспечивается их безопасность. Симметричный ключ действителен на протяжении всего сеанса, если только сеанс не превысил период действия ключа сеансу необходимо произвести микро-перезагрузку, как если бы произошел сбой соединения.

После того, как мост соединения становится безопасным, терминал отправляет запрос на доступ, который состоит из пароля пользователя. Если пароль правильный, терминал идентифицирует себя, и назначенный ему представитель ввода-вывода становится объектом сервера, отвечающим за соединение. Если с момента закрытия последнего сеанса произошли какие-либо структурные изменения в соответствующих устройствах ввода-вывода терминала, представитель ввода-вывода преобразуется в соответствии с ними.

Терминал и его представитель ввода-вывода имеют две односторонние очереди для всех входящих и исходящих данных соответственно, которые воплощают одну воображаемую двустороннюю очередь. Когда данные попадают в очередь входящих данных, они расшифровываются, и теперь, когда известно, как их следует обработать, они отправляются программной сущности-получателю. Когда данные поступают в исходящую очередь, они шифруются и отправляются как можно скорее в порядке приоритета. Стоит отметить, что эти абстрактные очереди не обязательно должны быть имплементированы как реальная очередь. Например, они могут быть асинхронными параллельно выполняющимися программами.

3.3. Механизм передачи данных устройств на сервер

Клиент-серверные системы в целом не несут в себе особой научной и практической новизны. Удалённое подключение к централизованным вычислительным мощностям тоже. Главной особенностью, преимуществом и сложностью облачной операционной системы типа «клиент-на-сервере» является универсальный и максимально абстрактный механизм передачи данных, генерируемых разнообразными физическими устройствами ввода, в серверную часть операционной системы в хорошо структурированном, единообразном и удобном к использованию системой и приложениями виде. Хотя отдельно взятый пользователь потребляет больше информации, чем генерирует, вывод данных пользователю из системы намного однообразнее и проще по структуре. Поэтому, разработке вышеупомянутого механизма передачи данных ввода была посвящена большая часть времени исследования.

Целью этого раздела является объяснение того, как данные передаются от пользователя до клиента-на-сервере (Client-on-the-Server, CotS). Вся концепция ввода основана на том, что каждое независимое

устройство ввода внутри терминала может при помощи терминала отправить через канал send-запрос на сервер.

Прежде всего, определим объекты передачи данных. Пакет состоит из полезных данных и заголовка. В заголовке хранится информация, обозначающая пункт назначения, в который данные должны быть отправлены, и описывающая метод их рассмотрения. Посылка (bale, databale) — это пакет, содержащий независимый и самодостаточный фрагмент данных. Поток данных, или просто поток (stream, datastream), используется для передачи последовательности зависимых посылок в режиме реального времени в строго определенном порядке. Такие посылки мало чем отличаются от обычных, поэтому отличия будем называть их кадрами потока (frames of the stream). Термин «датаграмма» может подразумевать как посылку, так и кадр потока.

Каждый send-запрос (send query) (Рис. 3.3.1) представляет собой пакет, обычно содержащий датаграмму. Его заголовок состоит из следующих полей: идентификатор канала, на который он отправляется, событие, в ответ на которое был сформирован запрос, и временную метку, состоящую из двух целых чисел, обозначающих значение времени и его смещение. Он также имеет следующие биты, обозначающие конфигурацию запроса: бит данных ввода, описывающий, содержит ли датаграмма данные ввода или внутрисистемное сообщение, биты окклюзии и ошибки, предназначение которых будет описано позже, и бит отметки времени, который устанавливается, когда метка времени определена и включена в заголовок. Биты окклюзии и ошибки должны быть отключены, если бит данных ввода установлен, но их можно установить одновременно. Раздел данных пакета либо содержит полезную нагрузку, либо пуст.

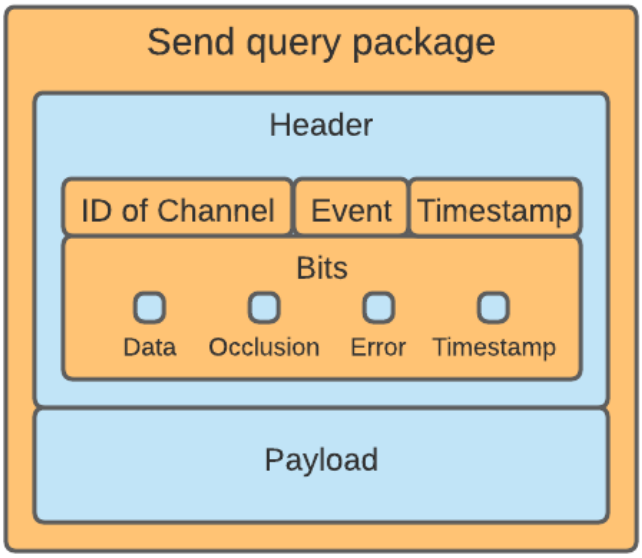

Рис. 3.3.1. Структура send-запроса

Каждое устройство ввода терминала подключено к объекту канала, связанному с ним. Канал состоит из двух частей, разделенных Интернетом: канала терминала и канала представителя ввода-вывода, причем первый концептуально представляет устройство ввода, а второй представляет цифровое отображение связанного устройства серверу и его приложениям. Таким образом, это абстрактная сущность, одни из потоков выполнения сегмента соединения сокета. Обе части канала обладают атрибутами и всей информацией об устройстве, необходимой для выполнения следующих задач: представлять устройство (уникальный идентификатор устройства и его тип), анализировать его действия (поддерживающиеся события), сжимать (распаковывать) и форматировать данные, отправляемые устройством, в зависимости от самих данных, а также для предоставления любой другой незначительной специфичной информации, которая может быть запрошена.

Атрибут «тип» привязан к классу устройства ввода. Его следует интерпретировать как общее название устройства. Например, компьютерная мышь, ещё одна мышь другого производителя и тачпад привязываются к каналам одного типа, поскольку выполняют одну и ту же функцию и мало чем отличаются с точки зрения сервера. Всех из очень точно описывает тип «Манипулятор». Сенсорный экран также подходит для устройств этого типа, поскольку с точки зрения ввода он практически неотличим от тачпада. В отличие от манипуляторов, клавиатура и камера обязаны иметь каналы разных типов. Уникальный идентификатор устройства необходим для предотвращения дублирования идентичных устройств ввода, таких как две

компьютерные мыши, подключенные одновременно. На примере двух мышей легко видеть, почему нельзя предоставить один экземпляр канала всем однотипным устройствам терминала. Набор поддерживаемых событий помогает классифицировать действия устройства.

Рассуждая о типе манипуляторов, можно заметить, что тачпад понимает какие-то особые жесты пальцев, но для мыши это неверно, а значит, в их объединении имеется ошибка. Это ошибочное мнение, и для подтверждения правильности унификации необходимо ввести определение понятия «событие». Каждый send-запрос может произойти только (за некоторыми исключениями, когда бит входных данных отключен) как результат действия, совершенного внешним миром (включая пользователя), которое активировало устройство ввода, или как реакция на запрос представителя ввода-вывода на получение некоторых данных. Таким образом, существует два типа событий: события реакции на внешнее событие «on», такие как on_click, on_ambient_brightness_increase и т. д., и события, происходящие в ответ на «request» представителя ввода-вывода, такие как request_camera_shot, request_mouse_position и т. д. События используются для того, чтобы указать представителю ввода-вывода почему был создан send-запрос.

Как было сказано ранее, в ОС разрешены два типа входных данных: посылки и потоки данных. Для потоков зарезервированы отдельные события. Перед идентификатором каждого события, подразумевающего поток, стоит префикс «s_» (например, s_request_camera для получения видеопотока). Для каждого потокового события внутри канала может быть открыт только один активный поток. Send-запрос может передавать только один кадр за раз. С того момента, как в канал терминала поступил первый кадр потока, поток считается активным и для его нужд запрашивается буфер (выделяемый из пула буферов). Все последующие кадры будут добавляться в буфер пока поток не будет закрыт. Закрытие (окклюзия) потока будет описываться позже. При отправке обычных полезных данных необходимо установить бит входных данных в заголовке send-запроса.

Каждые поток и посылка получают относительный приоритет, получаемый в зависимости от события. Это может ускорить их прохождение через системные очереди как терминала, так и представителя ввода-вывода, если они имеют более высокий приоритет. Посылки также обладают приоритетом локальной очереди, устроенным таким же образом. Каждый

кадр наследует свой приоритет от родительского потока. Если приоритет двух датаграмм одинаков, они обрабатываются в порядке FIFO.

Как упоминалось ранее, каждая посылка и каждый кадр отправляются в канал терминала посредством send-запроса. Там они классифицируются. Все посылки размещаются в очереди посылок (queue of bales), где они сортируются в соответствии с приоритетом локальной очереди. Затем они последовательно извлекаются, форматируются и сжимаются независимо друг от друга и в конечном итоге помещаются в системную исходящую очередь.

Кадры классифицируются в зависимости от присвоенного им события и отправляются в соответствующий буфер. Канал терминала должен передавать кадры буфера в системную исходящую очередь в порядке FIFO блоками или отдельно друг от друга. Каждый буфер выполняет свою собственную операцию форматирования и сжатия, поскольку при выполнении операции над одним из кадров может возникнуть необходимость учитывать соседние (предыдущие или последующие) кадры. Примером может служить простое поразрядное вычитание между двумя кадрами.

Когда пакет send-запроса проходит через канал терминала, его заголовок остается нетронутым. Только сегмент данных подвергается операциям форматирования и сжатия, а затем пакет передается в системную исходящую очередь. Операция форматирования означает любое необходимое изменение формы представления данных. В качестве абстрактного примера можно привести кодирование данных в формат Base64, хотя причина конкретно этого преобразования неясна с точки зрения концепции операционной системы типа «клиент-на-сервере». Общая структура канала терминала представлена на Рис. 3.3.2.

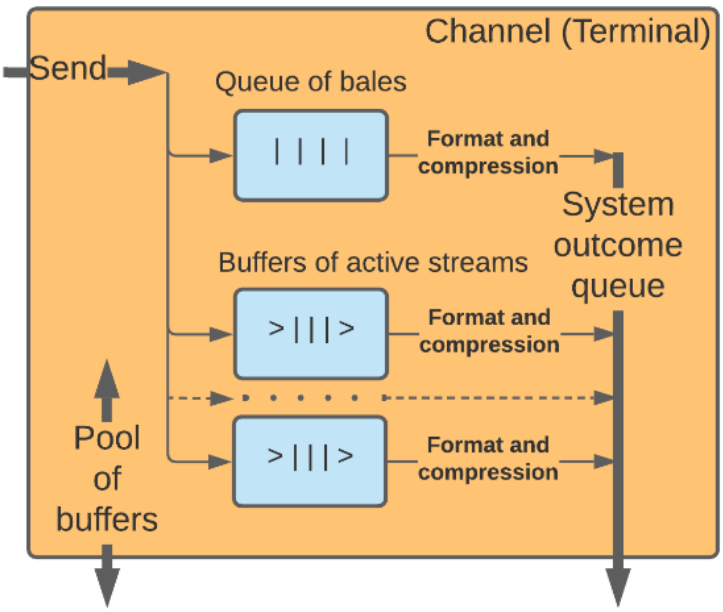

Рис. 3.3.2. Структура канала терминала

Существует два типа запросов, которые представитель ввода-вывода может подать на терминал: запрос ввода и запрос окклюзии. Все запросы обрабатываются процессором запросов терминала (request processor), который отправляет внутренние команды устройствам ввода. Запрос состоит из пакета, который содержит только заголовок. Заголовок включает в себя следующие параметры: бит типа, обозначающий один из двух типов запроса, идентификатор канала, на который отправляется запрос, и событие, несущее разное значение в зависимости от типа запроса. Кроме того, существует параметр временной метки, используемый только запросами ввода.

В первую очередь, процессор запросов терминала дешифрует запрос. Затем определяется тип запроса и запрос начинает обрабатываться по сути. Запросы ввода поддерживают только события «request_» и «s_request_». Несмотря на то, что устройство ввода, а не его канал терминала, производит данные по запросу, представитель ввода-вывода знает только о канале. Поскольку устройство ввода взаимно-однозначно соответствует каналу, мы можем направить запрос на драйвер устройства напрямую, без использования канала терминала в качестве медиатора.

Далее устройство подготавливает (генерирует) запрошенные данные и отправляет один send-запрос если была запрошена посылка или серию send-запросов для потоков. На всех пакетах параметр событий устанавливается таким же, как на пакете запроса. Как упоминалось выше,

запрос ввода имеет временную метку. Для посылок, созданных по запросу, или для первого кадра созданного потока, должен быть установлен бит временной метки, а значение поля временной метки в заголовке должно совпадать со значением в запросе. Также должен в заголовке send-запроса должен быть установлен бит данных ввода.

Запрос окклюзии (occlusion request) может быть направлен только на события «s_». Аналогичным образом мы извлекаем идентификатор устройства из параметра канала в заголовке запроса окклюзии и сообщаем устройству о необходимости закрыть (остановить) поток, соответствующий событию из запроса, если он открыт. После того, как мы остановим генерацию кадров потока, внутри канала терминала все еще останется связанный буфер, который, может быть, не пуст и его придется освободить. Для этого генерируется send-запрос с отключенным битом входных данных (пакет окклюзии не предназначен для использования приложениями), установленным битом окклюзии, параметрами «канал» и «событие» идентичными параметрам запроса окклюзии, и пустым сегментом данных. Запрос приводит к тому, что канал терминала обнуляет буфер и передает его обратно в пул буферов. Общая структура системы ввода терминала представлена на Рис. 3.3.3.

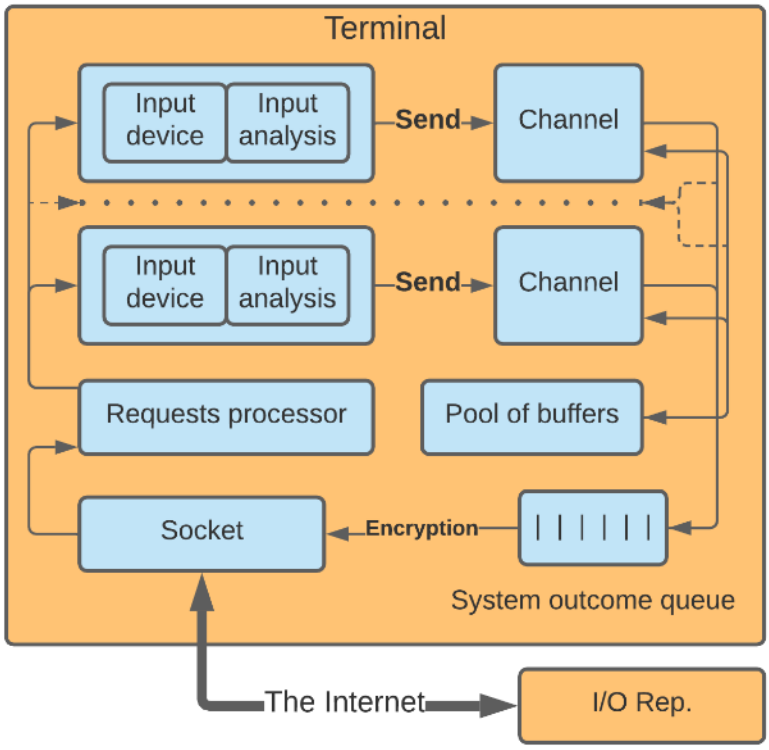

Рис. 3.3.3. Структура системы ввода терминала

Прежде чем явно уничтожить буфер, канал терминала отправляет пакет окклюзии в системную исходящую очередь и, следовательно, в представитель ввода-вывода. Этот шаг кажется слишком избыточным, но это не так.

Стоит разрешить устройству ввода самостоятельно генерировать send-запросы на окклюзию в случае, если такая функциональность будет сочтена полезной или если будет обнаружена критическая ошибка времени выполнения (runtime error). Если канал представителя ввода-вывода инициировал окклюзию и получил пакет, подтверждающий закрытие, то это избыточная, но полезная часть данных. С другой стороны, если устройство самостоятельно закрыло поток, для представителя ввода-вывода это удобный и эффективный способ осознать этот факт и проинформировать об этом приложения, которые раньше читали данные из потока. Поскольку мы оставили тело send-запроса пустым, то в случае окклюзии из-за неисправности устройства мы можем включить в запрос описание ошибки, установить бит ошибки и таким образом проинформировать представителя ввода-вывода. Если это может как-то помочь обработке ошибки, можно установить бит временной метки, а само значение временной метки в заголовке пакета установить на любое нужное значение.

Таким образом, представитель ввода-вывода уведомляется об ошибках, связанных с генерацией потоков, но не об ошибках, связанных с генерацией посылок. Поэтому необходимо разработать механизм для обработки таких ошибок, а также глобальный механизм обработки ошибок устройства ввода. Send-запрос со снятыми битами данных ввода и окклюзии, установленным битом ошибки, установленными при необходимости битом и сегментом метки времени, а также разделом данных, описывающим ошибку, представляет собой оптимальное решение. Все пакеты с отключенным битом данных ввода передаются в системную исходящую очередь через канал терминала в неизмененном виде, если у них отключен бит ошибки, или сжатыми универсальным образом, но неформатированными в противном случае.

Все входящие пакеты данных в конечном итоге появляются в расшифрованном виде во входящей очереди представителя ввода-вывода. Оттуда они отправляются в канал представителя ввода-вывода, чей аналог в терминале создал пакет. Каждый канал представителя ввода-вывода обменивается буферами с локальным пулом буферов, который является отдельным объектом от пула буферов терминала.

Также каналы представителя ввода-вывода могут генерировать запросы ввода и окклюзии в форме {ID канала, Событие} и адресовать их процессору запросов (request processor) представителя ввода-вывода. Последний должен сформировать пакет на основе запроса запроса для передачи своему аналогу в терминале. Структура пакета уже описана выше. Стоит отметить, что для запросов ввода сегмент временной метки пакета заполняется. Затем пакет шифруется и отправляется на Терминал через мост. Общая структура представителя ввода-вывода схематично представлена на Рис. 3.3.4.

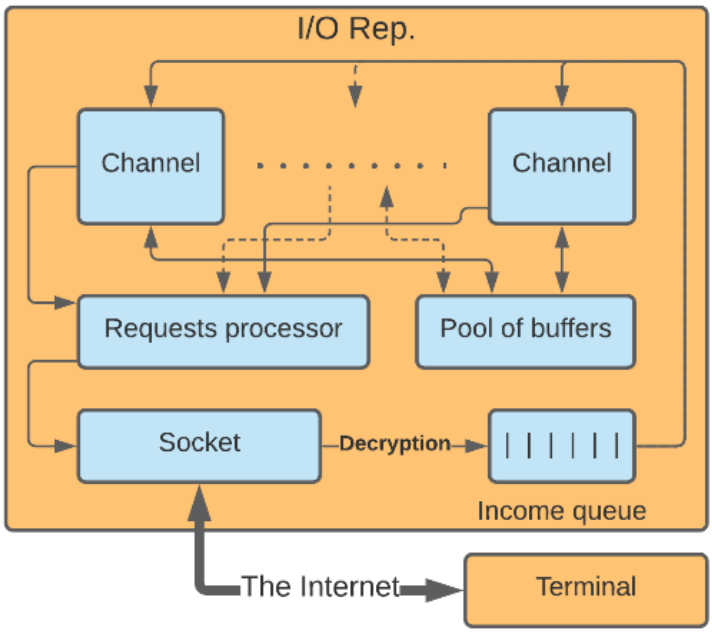

Рис. 3.3.4. Общая структура представителя ввода-вывода

Входные данные всегда доступны только для чтения. Таким образом, когда данные передаются от представителя ввода-вывода к приложениям, совершенно нет необходимости передавать данные по значению, а не как указатель, защищенный от модификации. Поскольку обычно одни и те же данные ожидают несколько приложений, а общие соображения требуют освобождения неиспользуемой памяти, предлагается следующая стратегия управления памятью.

Канал представителя ввода-вывода имеет два способа хранения данных в памяти: набор буферов для активных потоков и память посылок. Когда датаграмма поступает в канал представителя ввода-вывода, она классифицируется как посылка или кадр. Посылка распаковывается, форматируется и, в конечном итоге, помещается в очередь посылок, где она

находится до тех пор, пока из пула буферов в память посылок не будет выделен буфер одного элемента, размером с раздел данных посылки. Данные передаются в этот сегмент памяти, и на него генерируется интеллектуальный указатель. Интеллектуальный указатель — это указатель, который самостоятельно обнуляет и освобождает свой буфер, когда данные, на которые он указывает, больше не используются и больше не понадобятся. Посылки обрабатываются в порядке приоритета их локальной очереди. Как только умный указатель создан, пробуждающий вызов (awakening call), содержащий указатель, отправляется как в параллельный список обработчиков (list of handlers), связанный с событием посылки, так и в последовательный. Затем посылка удаляется из очереди. Если не было обработчиков, прослушивающих событие посылки, пробуждающий вызов не вызывает никакой активности.

При появлении первого кадра потока в канале представителя ввода-вывода для нужд потока выделяются два буфера из пула буферов. Буферы обнуляются и освобождаются, когда запрос на отправку окклюзии с их событием достигает терминала. Запрос окклюзии генерируется каналом представителя ввода-вывода, если все обработчики, которые раньше извлекали данные из потока, явно сигнализировали, что они им больше не нужны, или если для прослушивания события потока не зарегистрировано ни одного обработчика. По мере поступления в канал терминала все кадры классифицируются по своим событиям. Сначала все кадры потока освобождаются от заголовка и помещаются в буфер преобразования, в котором они находятся до тех пор, пока не будут зависимо или независимо друг от друга распакованы и отформатированы. Затем они поступают в активный буфер, где остаются до тех пор, пока используются обработчиками. Представитель ввода-вывода может однозначно назвать кадр неиспользуемым и удалить его, как только будут выполнены следующие критерии:

- он был запрошен всеми обработчиками, прослушивающими его событие,
- все обработчики, которые когда-либо запрашивали его, теперь используют другой, более новый кадр или закрыли поток, тем самым отказавшись от дальнейших запросов кадров,
- все кадры, пришедшие раньше него, удалены,
- но этот кадр не является последним поступившим в поток.

Как только первый кадр помещается в активный буфер, в список обработчиков соответствующего события поступает пробуждающий вызов со ссылкой на буфер.

Следует отметить, что в случае самостоятельного закрытия потока обработчики могут продолжить извлечение сохраненных в буфере фрагментов информации. Следовательно, если все еще присутствуют обработчики, которые еще не освободили свой поток, и поступает send-запрос окклюзии, их активный буфер сохраняется во временной памяти, пока они ее используют, и не освобождается. Обработчики явно не уведомляются о том, что их буфер перемещён во временную память. Эти решения оптимальны для событий «s_on_», которые имеют три варианта поведения в случае, если событие реального мира запускает их снова, пока поток все еще активен и не закрыт: игнорировать событие, интегрировать данные из нового события в тот же поток или самостоятельно закрыть поток и запустить его заново. В последнем случае обработчики продолжат извлекать устаревшие сохраненные данные, если захотят, и одновременно другие экземпляры тех же обработчиков будут задействованы пробуждающим вызовом из нового потока. Как мы увидим, обработчики потоков всегда поддерживают параллельное выполнение, поэтому экземпляры будут взаимодействовать корректно.

Пакеты с установленным битом ошибки обрабатываются каналом представителя ввода-вывода. Для этого в его архитектуру добавляется процессор ошибок. Общая структура канала представителя ввода-вывода представлена на Рис. 3.3.5.

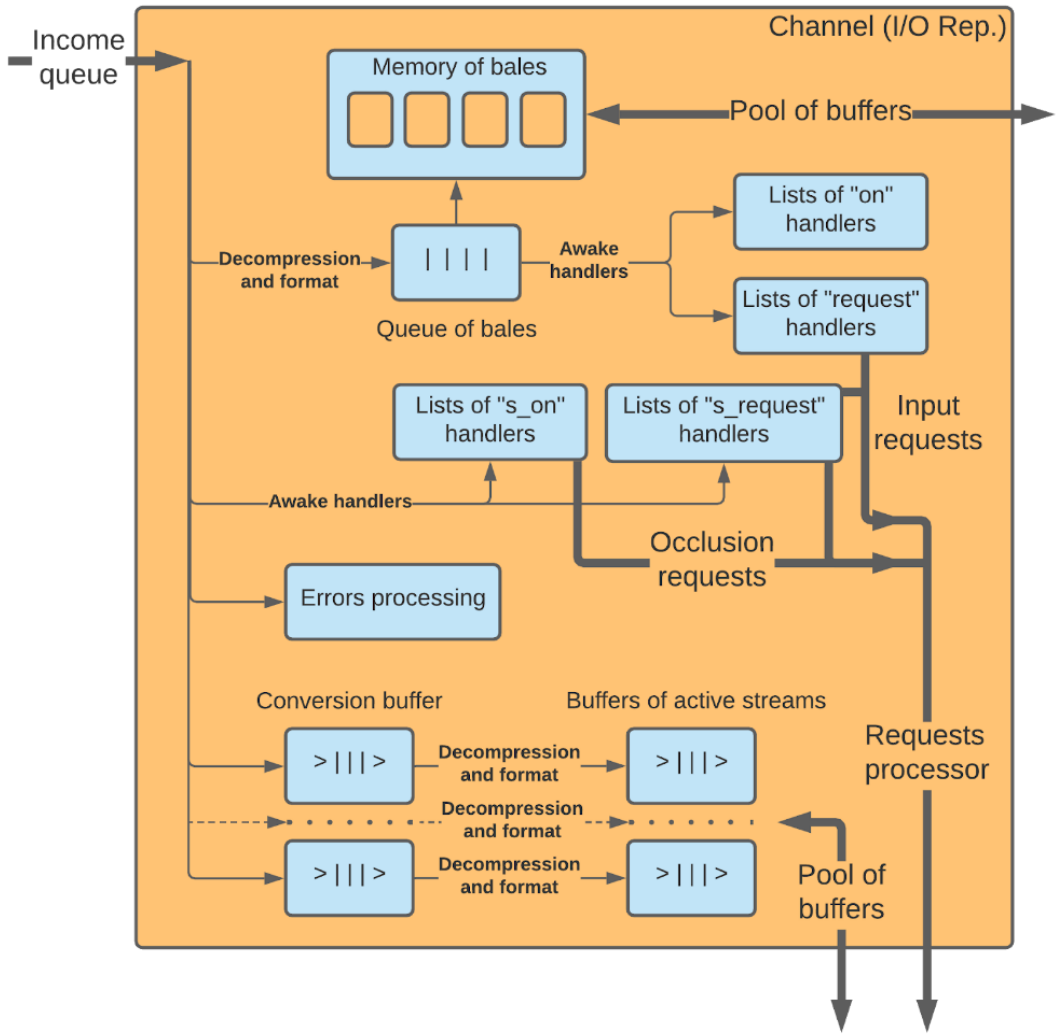

Рис. 3.3.5. Общая структура канала представителя ввода-вывода

3.4. Демонстрация функционирования имплементированных элементов

Проток связи не является наглядным объектом для человека. Однако, нам необходимо наглядно продемонстрировать, что имплементированные нами компоненты части протокола PODCAST, отвечающие за ввод данных, функционируют в целом и способны эффективно решать возложенные на него задачи. Выделим самые распространённые и популярные способы ввода данных пользовательского устройства: при помощи манипулятора (мыши) и клавиатуры. Визуализируем результаты процесса передачи сгенерированных ими данных через реализованные компоненты протокола соединения.

Будем захватывать данные о положении мыши и нажатии кнопок клавиатуры при помощи библиотеки Windows SDK [15] на тестовой машине под управлением ОС Windows. Данные захватываются в рамках каналов

терминала keyboard и mouse. Send-запросы с полученными данными маркируются двумя событиями: s_on_mouse_move и on_keypress, соответственно. Поле полезной нагрузки send-запроса мыши содержит её координаты относительно экрана тестовой машины. Поле send-запроса on_keypress содержит значение нажатого на клавиатуре сервера. Код канала терминала, генерирующего данные мыши, представлен в Приложении 1.

Далее, данные собираются в очередь основным процессом терминала. После этого они отправляются на сервер. Сервер запущен локально в рамках той же машины. Сокетная коммуникация происходит по localhost [16]. В потоке сервера, отображающем логическую сущность представителя ввода-вывода, данные разделяются по признаку событий и подготавливаются к передаче в графический интерфейс, который является наглядным для человека. Все реализованные нами компоненты операционной системы типа «клиент-на-сервере» написаны на языке C++. Этот язык известен тем, что реализация на нём примитивного графического интерфейса требует определённых временных и технических затрат. Напротив, реализовать веб-интерфейс в стеке JavaScript-HTML-CSS можно намного быстрее и с меньшими затратами [17]. Поэтому, было принято решение концептуально отделить реализацию графического интерфейса от реализации системы и протокола.

Принятие такого решения поднимает вопрос о коммуникации между программами, написанными на C++ и клиентской версии JavaScript. Необходимо перенаправить поток информации из компилируемого языка в интерпретируемый, опять же, без значительных затрат, потому что графическая компонента на данном этапе разработки не представляет технологической ценности. Поэтому, было решено использовать для цели перенаправления потока часто перезаписываемый файл в файловой системе тестовой машины. C++ без сложности будет производить запись файла при помощи стандартной библиотеки <fstream>. JavaScript-интерпретатор не имеет доступа к файловой системе из соображений безопасности. Значит, следует реализовать минимальный веб-сервер, который будет по запросу клиента возвращать содержимое файла на момент запроса. Лучше всего для этих целей подходит веб-сервер на NodeJS, который не требует предварительной настройки [18]. Код сервера на NodeJS, реализованный всего в несколько строк, представлен в Приложении 7. Клиентский JavaScript будет выполнять Ajax-запросы к веб-серверу. Поэтому, для

избежания проблемы CORS [19], response’ы сервера имеют заголовок Access-Control-Allow-Origin.

Составим графический веб-интерфейс следующим образом (Рис. 3.5.1). Большую часть интерфейса будет занимать табличная сетка, заполненная разноцветными прямоугольниками. Поверх неё будет двигаться красный курсор, повторяющий движения оригинального курсора в масштабе. Сетка прямоугольников нужна для визуальной привлекательности и для лучшего понимания масштаба. Прямоугольник, над которым «завис» курсор, динамично меняет свой цвет. В оставшейся части экрана расположено текстовое поле, которое заполняется по мере получения данных о нажатиях кнопок клавиатуры. Код веб-интерфейса на стеке JavaScript-HTML-CSS представлен в Приложении 8.

Рис. 3.5.1. Структура веб-интерфейса

Демонстрацию будем проводить таким образом (Рис 3.5.2), чтобы разница между оригинальным и моделируемым курсором была наглядной. Расположим окно браузера с открытым веб-интерфейсом так, чтобы оно занимало только часть (примерно четверть) экрана. Будем перемещать реальный курсор мыши в рамках всего экрана, и обратим внимание на его представителя в веб-версии, перемещающегося в масштабе экрана. Если бы мы пренебрегли масштабом и использовали для демонстрации полноэкранное окно веб-интерфейса, то расположение двух курсоров

совпадало бы. Откроем экранную клавиатуру. Будем использовать её, а не основную клавиатуру из следующих соображений: во-первых, на записях экрана видно, какие кнопки клавиатуры были нажаты, во-вторых, мы проверяем способность протокола одновременно (в категориях человека) передавать информации и о состоянии мыши, и о состоянии клавиатуры. Важно заметить, что поле для текста веб-интерфейса не предполагает ввода текста (реализовано при помощи тега) и не содержит курсора ввода. Это гарантирует, что данные в поле не вводятся непосредственно с клавиатуры, а проходят через протокол PODCAST.

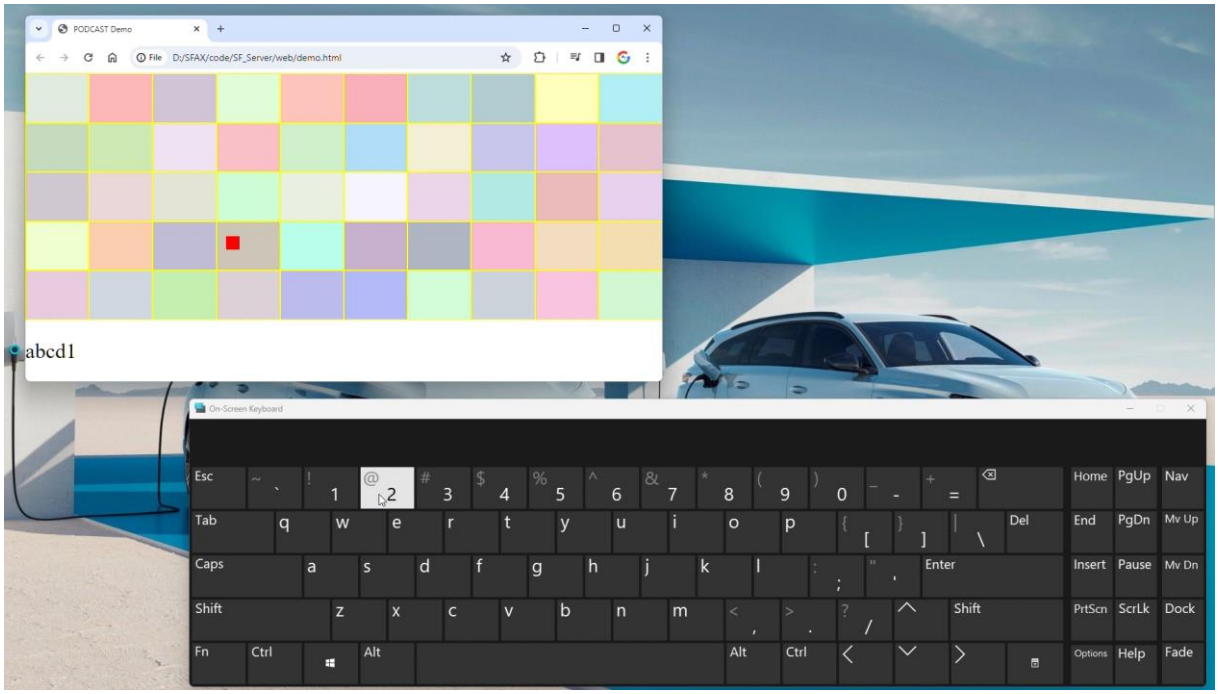

Рис. 3.5.2. Один из этапов демонстрации

В процессе демонстрации было выявлено, что все предполагаемые функции демонстрации выполняются в полном объёме. Задержка в движении курсора человеком незаметна, текст вводится с клавиатуры тоже без ощутимой задержки. Это значит, что концепция разработанного нами протокола связи доказана. Даже с учётом не совсем корректного способа передачи потока данных через файл, задержки не существенны. Следовательно, ограничения к практическому построению операционной системы типа «клиент-на-сервере», как минимум, носят технический, а не концептуальный характер, и связаны разве что со скоростями передачи данных по каналам компьютерных сетей.

Заключение

В ходе научной работы были проанализированы современные пользовательские операционные системы и были выделены их преимущества и недостатки. К преимуществам были отнесены удобство интерфейса, обширное число доступных приложений и аппаратных модулей, поддержка и обновление систем разработчиками. К недостаткам – ряд проблем безопасности, аппаратную зависимость и высокие системные требования. Отсутствие глубокой синхронизации было выделено как особый недостаток.

На основании проведённого анализа и после изучения современных частичных решений, устраняющих некоторые из недостатков, было принято решение о необходимости разработки особого класса облачных систем. Для обозначения этого класса был введён термин «клиент-на-сервере». Такой термин предполагает определённую модификацию клиент-серверной модели, в которой клиент выполняет только функции ввода-вывода и поддержания связи с сервером, а также имеет эмуляцию себя на сервере. Внедрение и имплементация экземпляра облачной операционной системы такого типа решает большинство проблем современных пользовательских операционных систем.

Нами была сформирована и определена фундаментальная архитектура операционной системы типа «клиент-на-сервере», а именно были выделены концептуальные компоненты (терминал, сервер, представитель ввода-вывода, пространство) и описаны основные типы взаимодействий между ними. Были отмечены некоторые аспекты и выделены основные принципы процесса фактической реализации такой операционной системы, принят ряд технических решений.

В ходе работы были проведено исследование, касающееся имплементации компонентов протокола связи, являющегося ключевым и наиболее научно ценным элементом облачной операционной системы типа «клиент-на-сервере». Были определены основные технологии и средства, при помощи которых нами было предложено реализовать протокол связи. Одним из необходимых этапов разработки было подробное описание процесса рукопожатия и первичной установки соединения между терминалом и сервером.

Так как отдельно взятый пользователь потребляет больше информации, чем генерирует, вывод данных пользователю из системы намного однообразнее и проще по структуре. Поэтому, разработке

вышеупомянутого механизма передачи данных ввода была посвящена большая часть времени исследования. В рамках этого исследования на основании определения компонентов фундаментальной архитектуры операционной системы была разработана теоретическая модель программной инфраструктуры каждого элемента пути передачи данных ввода от момента их генерации физическим устройством ввода до момента представления полученных данных на сервере.

На основании теоретических наработок были имплементированы некоторые компоненты операционной системы и часть структур протокола связи. При помощи реализованных элементов организована демонстрация работы системы передачи минимальных данных ввода, которая иллюстрирует передачу данных о положении мыши и нажатии кнопок клавиатуры. По наглядным итогам демонстрации мы пришли к выводу, что на базе разработанных нами технологий возможно построение облачной операционной системы типа «клиент-на-сервере», удовлетворяющей установленным в рамках этой работы требованиям.

Реализовав такую систему в будущем в полном объёме, можно инициировать новый период «взрывного» развития как аппаратного, так и программного компонентов пользовательских устройств. Изменение в сторону большего удобства интерфейсов взаимодействия пользователя с всегда расширяющимся спектром своих устройств является основой и залогом развития научного, технического и промышленного потенциала современной цивилизации.

Список использованных источников

1. Галкин, Д. В. Эволюция пользовательских интерфейсов: от терминала к дополненной реальности / Д. В. Галкин, В. А. Сербин // Гуманитарная информатика. – 2013. – № 7. – С. 35-49. – EDN QCKPDN.
2. Рогачева, Н. В. Интернет вещей: обзор основных проблем и задач / Н. В. Рогачева // Languages in professional communication, 29 апреля 2021 года. – ООО «Издательский Дом «Ажур», 2021. – Р. 558-563. – EDN CRGRJQ.
3. Таненбаум Э., Бос Х. Современные операционные системы. 4-е изд. — СПб.: Питер, 2015. — 1120 с.
4. Столлингс, В. Операционные системы: внутренняя структура и принципы проектирования. 9-е изд. — СПб.: ООО «Диалектика», 2020. — 1264 с.
5. Абрамович, Е. Д. Операционная система Chrome OS или облачная операционная система / Е. Д. Абрамович // Инновационные технологии и образование : Материалы международной научно-практической конференции. В 2 частях, Минск, 28 апреля 2022 года. Том Часть 2. – Минск: Белорусский национальный технический университет, 2022. – С. 3-8. – EDN BZWXFM.
6. Chen, P. M. When virtual is better than real [operating system relocation to virtual machines] / P. M. Chen, B. D. Noble // Proceedings Eighth Workshop on Hot Topics in Operating Systems, Elmau, Germany, 2001. – С. 133-138. – DOI: 10.1109/HOTOS.2001.990073.
7. Лихтциндер, Б. Я. Дистанционный анализ трафика с помощью утилиты Team Viewer и программы WireShark / Б. Я. Лихтциндер, Л. А. Сарычев // Технические науки: проблемы и перспективы : Материалы V Международной научной конференции, Санкт-Петербург, 20–23 июля 2017 года. – Санкт-Петербург: Свое издательство, 2017. – С. 1-3. – EDN ZBLDCR.
8. Подход к обнаружению компьютерных атак на RDP при расследовании компьютерных инцидентов / В. В. Данилов, П. А. Романов, И. А. Бугаев, П. В. Тимашов // Защита информации. Инсайд. – 2020. – № 5(95). – С. 68-71. – EDN AKJBTX.
9. Гончаров, А. М. Размещение компонентов корпоративных информационных систем в облачных структурах / А. М. Гончаров // Актуальные проблемы и перспективы развития экономики : Труды Юбилейной XX Всероссийской с международным участием научно-практической конференции, Симферополь - Гурзуф, 11–13 ноября 2021

года. – Симферополь: Издательский дом КФУ, 2021. – С. 224. – EDN RRJRGF.
10. Макеев, А. Ю. Использование технологии 5G и особенности расчета скорости передачи данных / А. Ю. Макеев // Оригинальные исследования. – 2022. – Т. 12, № 9. – С. 59-65. – EDN AFUAVI.
11. Гончаров, А. М. О проблеме параллельной записи памяти в облачных операционных системах / А. М. Гончаров // Теория и практика экономики и предпринимательства : Труды XX Международной научно-практической конференции, Симферополь - Гурзуф, 20–22 апреля 2023 года / Под редакцией Н.В. Апатовой. – Симферополь: Крымский федеральный университет им. В.И. Вернадского, 2023. – С. 230-231. – EDN WAKLCR.
12. Бирюков, А. Криптоанализ американского стандарта шифрования AES-192 и AES-256 на основе связанных ключей / А. Бирюков, Д. Ховратович // Интеграл. – 2010. – № 1. – С. 35-37. – EDN MNJCPD.
13. Набиев, С. Р. Ассиметричные методы шифрования: RSA, ELGAMAL, ECC / С. Р. Набиев // Современные научные исследования и разработки. – 2018. – Т. 1, № 11(28). – С. 474-475. – EDN YULACL.
14. Гончаров, А. М. Методы защиты сокета Беркли от спам-атак до момента авторизации пользователя / А. М. Гончаров // Проблемы информационной безопасности социально-экономических систем : VIII Всероссийская с международным участием научно-практическая конференция, Симферополь - Гурзуф, 17–19 февраля 2022 года. – Симферополь: Крымский федеральный университет им. В.И. Вернадского, 2022. – С. 93. – EDN RONXXL.
15. Jana A. Kinect for Windows SDK Programming Guide. - Packt Pub., 2012. - 392 c.
16. Еровлева, Р. В. Ошибки при регистрации сервисов как localhost на Eureka / Р. В. Еровлева, П. А. Еровлев // Постулат. – 2021. – № 7(69). – EDN VLDGQV.
17. Potapov, V. A. Website development using HTML, CSS and Javascript / V. A. Potapov, D. A. Loshkareva // , 18–19 октября 2022 года, 2022. – P. 983-984. – EDN CKEXTO.
18. Сидорова, Е. В. Решение задачи построения графа зависимостей программных модулей в системе Node.js / Е. В. Сидорова, Н. Г. Дмитриева, Н. А. Калинина // Труды НГТУ им. Р.Е. Алексеева. – 2019. – № 4(127). – С. 44-52. – DOI 10.46960/1816-210X_2019_4_44. – EDN UBQSCA.

19. Вафин, И. И. Значение CORS для безопасности веб-приложений / И. И. Вафин // Совершенствование науки и образования в области естественных и технических исследований : Материалы XXXVI Всероссийской научно-практической конференции, Ставрополь, 21 декабря 2023 года. – Ставрополь: Общество с ограниченной ответственностью "Ставропольское издательство "Параграф", 2023. – С. 22-24. – EDN NOFTAK.

Приложение 1. Пример канала ввода терминала

```
#include <sstream>
#include <iostream>
#include <thread>
#include <Windows.h>

#include "../../include/utility_subject/Priority_queue.cpp"
#include "../../include/utility_subject/Pipe.cpp"
#include "../../include/utility_subject/Guid.cpp"
#include "../../include/Send_input.cpp"

using namespace std;
using namespace Io_global;

ostream& operator<<(ostream& os, const pair<int, int>& p) {
    return os << p.first << ' ' << p.second;
}

void mouse_reader(Utility::Simple_buffer<Send_input*>* buf) {
    while (true) {
        POINT p;
        GetCursorPos(&p);
        pair<int, int> coords{ p.x, p.y };
        ostringstream os;
        os << coords;
        Send_input* si = new Send_input{ {"s_on_mouse_move"}, os.str(), true,
false, false, false };
        buf->put(si);
        Sleep(1);
    }
}

void converter(Utility::Simple_buffer<Send_input*>* buf, Utility::Pipe_instance*
pp) {
    bool first = true;
    int last_x, last_y;
    while (true) {
        Send_input* si = buf->get();

        istringstream is{ si->payload };
        int x, y;
        int x_save, y_save;
        is >> x >> y;
        x_save = x, y_save = y;

        string res;

        if (!first)  {
            x -= last_x, y -= last_y;
        }
        if (first || x != 0 || y != 0) {
            res = string{ reinterpret_cast<char*>(&x), sizeof(int) };
            res += string{ reinterpret_cast<char*>(&y), sizeof(int) };

            si->payload = res;
            pp->send(si->to_string());

            first = false;
        }

        delete si;

        last_x = x_save, last_y = y_save;
    }
}

int main(int argc, char* argv[])
{
    Utility::GUID guid{ string{argv[1]} };
```

```
    Utility::Pipe_listener p{ string("sfax") + guid.str()};

    Utility::Simple_buffer<Send_input*> buf{ 512 };

    string ready = p.receive();

    thread readout{ mouse_reader, &buf };
    thread processer{ converter, &buf, &p };

    readout.join();
    processer.join();

    return 0;
}
```

Приложение 2. DNS сервер ОС типа «клиент-на-сервере»

```
#include "./Db.cpp"
#include "./nanodbc/nanodbc.cpp"

#include "../utility_subject/Socket.cpp"

#include <string>
#include <thread>
#include <memory>

using namespace std;

namespace DNS_device {
	class Server {
	public:
		Server(const string& db_uname, const string& db_psw, const string&
port);
		void activate();
		~Server() { if (sock) delete sock; }
	private:
		void handle(Utility::Socket_instance* inst);

		string port;
		Broker_device::DB db;
		Utility::Socket_server* sock;
	};

	Server::Server(const string& db_uname, const string& db_psw, const string&
port)
		:sock{ new Utility::Socket_server{port} }, db{ db_uname, db_psw },
port{ port }
	{

	}

	void Server::activate() {
		while (true) {
			try {
				Utility::Socket_instance* inst = new
Utility::Socket_instance{ sock->accept_client() };
				if (inst->is_valid()) {
					thread sock_thread{ &Server::handle, this, inst };
					sock_thread.detach();
				}
			}
			catch (Utility::Bad_socket&) {
				delete sock;
				sock = new Utility::Socket_server{ port };
			}
			catch (...) {

			}
		}
	}

	void Server::handle(Utility::Socket_instance* inst) {
		unique_ptr<Utility::Socket_instance> ptr{ inst };

		string uname = ptr->receive(500);
		if (uname == "") {
			ptr->send("not_found");
			return;
		}

		string query = string("select ip_address, port from [Container] where
user_name='") + uname + "'";
		nanodbc::result res;
		try { res = db.query(query); }
		catch (...) {
```

```
                        ptr->send("not_found");
                        return;
                }

                if (res.next()) {
                        string ans = res.get<string>(0, "") + ":" + res.get<string>(1,
"");
                        ptr->send(ans);
                }
                else
                        ptr->send("not_found");
        }
}
```

Приложение 3. Класс моста соединения Bridge

```
#include <string>

#include "../utility_subject/Socket.cpp"
#include "../utility_subject/Rsa.cpp"
#include "../utility_subject/Aes.cpp"

using namespace std;

namespace User_device {
       class Bad_bridge : public exception {
    public:
        Bad_bridge(const string& s, int error_code = 0) :err{ string("Terminal's
bridge: ") + s }, err_code{ error_code } {}
        const char* what() const override { return err.data(); }

        /*
        0 - not specified
             1 - RSA key not received
             2 - AES key transmission failed
             3 - password not transmited
             4 - password not authorized
             5 - bad RSA key
        */
        int error_code() const { return err_code; }
    private:
        string err;
        int err_code;
    };

       class Bridge : public Utility::Socket_client {
       public:
             Bridge(const string& server_ip, const string& server_port, const
string& password, unsigned int to_client_buf_sz, unsigned int to_server_buf_sz);
             bool send(const string& s) override;
             string receive(int timeout = -1) override;
       private:
             Utility::AES aes;
       };

       Bridge::Bridge(const string& server_ip, const string& server_port, const
string& password, unsigned int to_client_buf_sz, unsigned int to_server_buf_sz)
             :Socket_client{ to_client_buf_sz, to_server_buf_sz }
       {
             establish(server_ip, server_port);

             string public_key = Socket_client::receive();
             if (public_key == "") throw Bad_bridge("RSA key not received", 1);

             Utility::RSA_client rsa{ public_key };
             if (!rsa.valid()) throw Bad_bridge("bad RSA key", 5);

             string aes_key = rsa.encode(aes.get_key());
             bool success = Socket_client::send(aes_key);

             if (!success) throw Bad_bridge("AES key transmission failed", 2);
             for (char& c : aes_key) c = 0;

             success = send(password);
             if (!success) throw Bad_bridge("password not transmitted", 3);

             string psw_responce = receive();
             if (psw_responce != "granted") throw Bad_bridge("password not
authorized", 4);
       }

       bool Bridge::send(const string& s) {
             string cipher = aes.encode(s);
             return Socket_instance::send(cipher);
```

```
	}

	string Bridge::receive(int timeout) {
		string cipher = Socket_instance::receive(timeout);
		if (cipher == "") return cipher;
		else return aes.decode(cipher);
	}
}
```

Приложение 4. Класс системного супервайзера

```
#include "../db_broker_subject/Db.cpp"
#include "../utility_subject/Socket.cpp"
#include "../utility_subject/Rsa.cpp"
#include "../utility_subject/Json.cpp"

#include <string>
#include <memory>
#include <sstream>

using namespace std;

namespace Supervisor {

	class Supervisor {
	public:
		Supervisor(const string& db_login, const string& db_psw, const string&
port);
		void activate();
		~Supervisor();
	private:
		void handle(Utility::Socket_instance* sock);

		bool update_keys(const string& guid);

		Utility::Socket_server* sock;
		Broker_device::DB db;
		string port;
	};

	Supervisor::Supervisor(const string& db_login, const string& db_psw, const
string& port)
		:port{ port }, sock{ new Utility::Socket_server{ port } }, db{
db_login, db_psw }
	{

	}

	void Supervisor::activate() {
		while (true) {
			try {
				Utility::Socket_instance* inst = new
Utility::Socket_instance{ sock->accept_client() };
				if (inst->is_valid()) {
					thread sock_thread{ &Supervisor::handle, this,
inst };
					sock_thread.detach();
				}
			}
			catch (Utility::Bad_socket&) {
				delete sock;
				sock = new Utility::Socket_server{ port };
			}
			catch (...) {

			}
		}
	}

	void Supervisor::handle(Utility::Socket_instance* inst) {
		unique_ptr<Utility::Socket_instance> sock_ptr{ inst };
		string command_str = inst->receive(5000);
		if (command_str == "") return;
		istringstream is{ command_str };
		string command; is >> command;
		if (command == "new_keys") {
			string guid; is >> guid;
			if (update_keys(guid))
				inst->send("updated");
```

```
                        else
                                inst->send("bad");
                }
        }

        bool Supervisor::update_keys(const string& guid) {
                try {
                        Utility::RSA_server rsa{};
                        rsa.seek_keys();
                        Utility::Signature_signer sign{};
                        sign.seek_keys(3072);

                        string query = string() + "update [Container] set public_key='"
+ rsa.get_public_key()
                                + "', signature_key='" + sign.get_public_key() + "',
rsa_key_created=getdate(), "
                                "signature_key_created=getdate() where container_guid='"
+ guid + "'";
                        nanodbc::result res;
                        res = db.query(query);

                        if (res.has_affected_rows()) {
                                query = string() + "select path from [Container] where
container_guid='" + guid + "'";
                                res = db.query(query);
                                if (!res.next()) throw 0;

                                Utility::JSON json{ res.get<string>("path", "") +
"config.json" };

        json["private_key"].SetString(rsa.get_private_key().data(), json.alloc());

        json["signature_key"].SetString(sign.get_private_key().data(), json.alloc());
                                json.apply();
                        }
                }
                catch (...) {
                        return false;
                }
                return true;
        }

        Supervisor::~Supervisor() {
                if (sock) delete sock;
        }

}
```

Приложение 5. Передача хэша пароля терминала

```
string Broker::get_teminal_password(const string& guid, const string& command) {
		try {
			string query = string() + "select terminal_password from
[Container] where container_guid='" +
				guid + "'";
			auto res = database.query(query);
			if (!res.next()) throw 0;
			return res.get<string>("terminal_password", "");
		}
		catch (...) {
			return "";
		}
	}


string Db_rep::get_terminal_password() {
		bool good = false;
		string ans;
		while (!good) {
			command_mut.lock();
			try {
				if (!send("get_terminal_password")) throw 0;
				ans = receive(timeout);
				if (ans == "") throw 0;
				good = true;
			}
			catch (...) {
				reboot();
			}
			command_mut.unlock();
			if (!good) Utility::Delay{ 1000 };
		}

		return ans;
	}
```

Приложение 6. Класс брокера базы данных

```
#include "./Db.cpp"

#include "../utility_subject/Socket.cpp"
#include "../utility_subject/Rsa.cpp"
#include "../utility_subject/Aes.cpp"

#include <string>
#include <thread>
#include <memory>
#include <sstream>
#include <map>

using namespace std;

namespace Broker_device {

	class Broker {
	public:
		Broker(const string& db_uname, const string& db_password, const
string& port);
		void activate();
		~Broker() { if (sock) delete sock; }
	private:
		void handle(Utility::Socket_instance* inst);
		bool handshake(Utility::Socket_instance* inst, Utility::AES* aes,
string& guid);
		string analyse(const string& command, const string& guid);
		void commands_into_map();

		Utility::Socket_server* sock;
		string port;
		DB database;
		const int wait_delay = 5000;
		const int wait_delay_long = 20000;

		map<string, string(Broker::*)(const string&, const string&)>
commands_map;
		//commands
		string get_teminal_password(const string& guid, const string&
command);
		string get_terminal_socket_port(const string& guid, const string&
command);
	};

	Broker::Broker(const string& db_uname, const string& db_password, const
string& port)
		:database{db_uname, db_password}, sock { new Utility::Socket_server{
port } }, port{ port }
	{
		commands_into_map();
	}

	void Broker::activate() {
		while (true) {
			try {
				Utility::Socket_instance* inst = new
Utility::Socket_instance{ sock->accept_client() };
				if (inst->is_valid()) {
					thread sock_thread{ &Broker::handle, this, inst };
					sock_thread.detach();
				}
			}
			catch (Utility::Bad_socket&) {
				delete sock;
				sock = new Utility::Socket_server{ port };
			}
			catch (...) {
```

```
                }
            }
        }

        bool Broker::handshake(Utility::Socket_instance* inst, Utility::AES* aes,
string& guid) {
            try {
                guid = inst->receive(wait_delay);
                if (guid == "") throw 0;

                string query = string("") + "select datediff(day,
rsa_key_created, getdate()) as rsa_diff,"
                    " datediff(day, signature_key_created, getdate()) as
sign_diff, public_key, signature_key"
                    " from [Container] where container_guid='" + guid + "'";
                nanodbc::result res = database.query(query);
                if (!res.next()) throw 0;

                string public_key = res.get<string>("public_key", "");
                string sign_key = res.get<string>("signature_key", "");
                if (!inst->send(public_key)) throw 0;
                if (!inst->send(sign_key)) throw 0;

                string keys_validation = inst->receive(wait_delay);
                if (keys_validation != "good_keys") throw 0;

                string signature = inst->receive(wait_delay);
                if (signature == "") throw 0;
                Utility::Signature_verifier signature_verifier{ sign_key };
                if (!signature_verifier.verify(guid, signature)) throw 0;

                Utility::RSA_client rsa{ public_key };
                string aes_key = aes->get_key();
                string aes_to_send = rsa.encode(aes_key);
                if (!inst->send(aes_to_send)) throw 0;

                string update = "  ";
                update[0] = (res.get<int>("rsa_diff") > 360) ? 1 : 0;
                update[1] = (res.get<int>("sign_diff") > 360) ? 1 : 0;
                if (!inst->send(aes->encode(update))) throw 0;

                if (update[0]) {
                    string rsa_new = aes->decode(inst-
>receive(wait_delay_long));
                    Utility::RSA_client rsa_updated{ rsa_new };
                    if (rsa_new == "" || !rsa_updated.valid()) throw 0;
                    query = string("") + "update [Container] set
public_key='" + rsa_new +
                        "', rsa_key_created=getdate() where
container_guid='" + guid + "'";
                    res = database.query(query);
                    if (!res.has_affected_rows()) throw 0;
                }
                if (update[1]) {
                    string sign_new = aes->decode(inst-
>receive(wait_delay_long));
                    Utility::Signature_verifier sign_updated{ sign_new };
                    if (sign_new == "" || !sign_updated.valid()) throw 0;
                    query = string("") + "update [Container] set
signature_key='" + sign_new +
                        "', signature_key_created=getdate() where
container_guid='" + guid + "'";
                    res = database.query(query);
                    if (!res.has_affected_rows()) throw 0;
                }
            }
            catch (...) {
                if (inst->is_valid()) inst->send(string{ char(0), 1 });
                return false;
            }
```

```
		return true;
	}

	void Broker::handle(Utility::Socket_instance* inst) {
		unique_ptr<Utility::Socket_instance> sock_ptr{ inst };
		unique_ptr<Utility::AES> aes_ptr(new Utility::AES);
		string guid;

		if (!handshake(inst, aes_ptr.get(), guid)) return;

		while (true) {
			try {
				string command = inst->receive();
				if (command == "" && inst->is_closed()) break;
				try { command = aes_ptr->decode(command); }
				catch (...) { command = ""; }
				if (command == "close") break;
				inst->send(aes_ptr->encode(analyse(command, guid)));
			}
			catch (...) {

			}
		}
	}

	string Broker::analyse(const string& s, const string& guid) {
		try {
			istringstream is{ s };
			string command;
			is >> command;
			auto it = commands_map.find(command);
			if (it == commands_map.end()) throw 0;
			string(Broker::*func_ptr)(const string&, const string&) = it-
>second;
			string arguments = s.substr(command.size(), s.size() -
command.size());
			return (this->*(func_ptr))(guid, guid);
		}
		catch (...) {
			return "";
		}
	}
}

#include "./Broker_commands.cpp"
```

Приложение 7. Веб-сервер на NodeJS

```
var http = require('http');
var fs = require('fs');
var filePath = "D:\\SFAX\\code\\SF_Server\\SF_Server\\output.txt";

http.createServer(function (req, response) {

  fs.readFile(filePath, {encoding: 'utf-8'}, function(err,data){
    if (!err) {
        response.writeHead(200, {'Content-Type': 'text/html', 'Access-Control-
Allow-Origin': '*'});
        response.write(data);
        response.end();
    } else {
        response.writeHead(200, {'Content-Type': 'text/html', 'Access-Control-
Allow-Origin': '*'});
        response.write('');
        response.end();
    }
  });
}).listen(8020);
```

Приложение 8. Веб-интерфейс демонстрации

```
<html>
    <head>
        <title>PODCAST Demo</title>
        <style>
            body {
                margin: 0;
            }

            .blocks {
                position: absolute;
                width: 100%;
                height: 80%;
                top:0;
            }

            .text {
                position: absolute;
                width: 100%;
                height: 20%;
                top:80%;
            }

            .row {
                position: relative;
                width: 100%;

            }

            .cell {
                position: relative;
                display: inline-block;
                border: yellow 1px solid;

            }

            .text {
                font-size: xx-large;
                display: flex;
                flex-direction: row;
                align-items: center;
            }

            .cursor {
                position: absolute;
                width: 20px;
                height: 20px;
                background-color: red;
                top:0; left: 0;
            }
        </style>
    </head>
    <body onload="main()">
        <div class="blocks">

        </div>
        <div class="text">
        </div>

        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
        <script src="./logic.js?v=2"></script>
    </body>
</html>
```

LOGIC.JS:

```
var blocks = document.getElementsByClassName('blocks')[0];
```

```
var bh, bw;
var size_x = 10, size_y = 5;
var c =80;

function main() {

    var b_text = "";

    b_h = blocks.offsetHeight-size_y*2;
    b_w = blocks.offsetWidth-size_x*2;
    for (var i = 0; i < size_y; i++) {
        b_text += "<div class='row'>";
        for (var j = 0; j < size_x; j++) {
            b_text += "<div class='cell' style='background-color: rgb(" + (255-
Math.floor(Math.random() * c))  + ", " + (255-Math.floor(Math.random() * c))  + ",
" + (255-Math.floor(Math.random() * c))  + "); width:" + b_w / size_x + "px;
height:"+ b_h / size_y +"px'></div>";
        }
        b_text += "</div>"
    }
    b_text += "<div class='cursor'></div>"
    blocks.innerHTML = b_text;

    request();
}

function request() {
    $.ajax({
        url: 'http://localhost:8020',
        type: "GET",
        success: function (data) {
            display(data)
        }
    });

    setTimeout(request, 100);
}

function display(data) {
    if (data.indexOf(' ') == -1) {
        var s = document.getElementsByClassName("text")[0].innerHTML;
        if (s[s.length -1 ] !== data)
            document.getElementsByClassName("text")[0].innerHTML += data;
    } else {
        const regexp = /(\d+) (\d+)/g;
        const matches = data.matchAll(regexp);
        var i = 0;
        var x, y;
        for (match of matches) {
            x = match[1];
            y = match[2];
        }

        x = Math.floor(x / 1920 * b_w);
        y = Math.floor(y / 1080 * b_h);

        var cursor = document.getElementsByClassName("cursor")[0];
        cursor.style.left = x+"px";
        cursor.style.top = y+"px";

        var x_block = Math.floor (x / b_w * size_x);
        var y_block = Math.floor (y / b_h * size_y);

        document.getElementsByClassName("row")[y_block].getElementsByClassName("cel
l")[x_block].style.backgroundColor = "rgb(" + (255- Math.floor(Math.random() *
c))  + ", " + (255-Math.floor(Math.random() * c))  + ", " + (255-
Math.floor(Math.random() * c))  + ")";
    }
}
```