

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

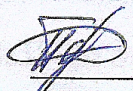
Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль подготовки: «Программное обеспечение средств вычислительной
техники и автоматизированных систем»

БАКАЛАВРСКАЯ РАБОТА

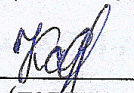
на тему

**ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СЕТИ
ПИРИНГОВЫХ УЗЛОВ ДЛЯ ВЫПОЛНЕНИЯ
РАСПРЕДЕЛЕННЫХ ПРИЛОЖЕНИЙ**

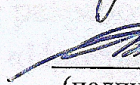
Студент

 31.05.22 Перфильев Д.П.
(подпись, дата)

Руководитель

 31.06.22 Карамышева Н.С.
(подпись, дата)

Нормоконтролёр

 31.05.22 А.С. Бычков
(подпись, дата)

Работа допущена к защите (протокол зас. кафедры от 02.06.2022 г. № 13)

Заведующий кафедрой

 М.А. Митрохин
(подпись)

Работа защищена с отметкой отлично (протокол заседания ГЭК от

15.06.2022 г. № 9)

Секретарь ГЭК


(подпись)

Е.И. Гурин

Пенза 2022

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
М.А. Митрохин
«26» апреля 2022г.

**ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ**

1. Студент Перфильев Дмитрий Павлович гр.18ВВ2 факультета вычислительной техники.
2. Тема работы: «Программная реализация сети пиринговых узлов для выполнения распределенных приложений»

Тема утверждена приказом ПГУ №357/о от "26" апреля 2022г.

3. Руководитель бакалаврской работы к.т.н., доцент, каф. ВТ ПГУ Карамышева Н.С.

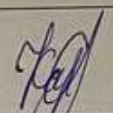
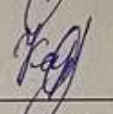
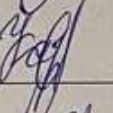
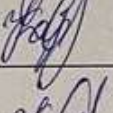
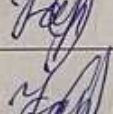
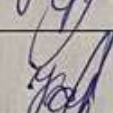
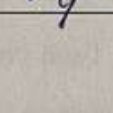
4. Задание на работу (назначение разработки, исходные данные и т.п.)

Разработать приложение «Программная реализация сети пиринговых узлов для выполнения распределенных приложений», позволяющее реализовывать схему взаимодействия компьютеров в сети согласно концептуальному графу распределенного приложения. Приложение должно решать такие задачи как, взаимодействие удаленных узлов в вычислительной сети, реализация любой схемы взаимодействия удаленных узлов между собой, возможность выполнения определенной задачи на конкретном узле.

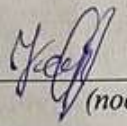
5. Перечень подлежащих разработке вопросов

Требуется произвести анализ предметных областей: изучить существующие технологии распределенных вычислений, выделить их достоинства и недостатки. Выбрать язык программирования, который подойдет для создания сетевого соединения между узлами сети и использования многопоточного программирования. Определить архитектуру приложения и вспомогательные средства разработки. Изучить функционал библиотек boost для реализации сетевого обмена данными и сериализации.

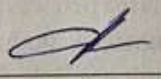
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Изучение предметной области	32	26.04.2022 – 28.05.2022	
2	Анализ требований	32	29.04.2022 – 30.05.2022	
3	Создание макета графического интерфейса	30	1.04.2022 – 3.05.2022	
4	Выбор технологий, применяемых при проектировании и разработке	40	4.05.2022 – 8.05.2022	
5	Реализация графического интерфейса	66	9.05.2022 – 15.05.2022	
6	Разработка основной части программы	70	16.05.2022 – 23.05.2022	
7	Оформление пояснительной записки	54	24.05.2022 – 31.05.2022	

Дата выдачи задания "26" апреля 2022 г.

Руководитель бакалаврской работы  26.04.22 Карамышева Н.С.
(подпись, дата)

Задание к исполнению принял студент  26.04.22 Перфильев Д.П.
(подпись, дата)

Работу к защите допустить
Декан факультета  Фионова Л.П.
(подпись, дата)
09.06.2022

Реферат

Бакалаврская работа содержит 111 листов, 17 рисунков, 2 таблицы, 8 источников, 3 приложения.

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ СЕТИ ПИРИНГОВЫХ УЗЛОВ ДЛЯ ВЫПОЛНЕНИЯ РАСПРЕДЕЛЕННЫХ ПРИЛОЖЕНИЙ, КОНЦЕПТУАЛЬНЫЙ ГРАФ РАСПРЕДЕЛЕННОГО АЛГОРИТМА, BOOST, P2P, TCP

Цель работы – разработать программу, позволяющую выполнять распределенные приложения в сети пиринговых узлов.

Приложение должно позволять реализовывать любую схему взаимодействия компьютеров в сети согласно концептуальному графу распределенного приложения при использовании всех дейкстровских конструкций структурного программирования.

Разработка проводилась в среде программирования *Visual Studio 2019*. В качестве языка программирования был выбран C++.

В результате проведенной работы разработано приложение, удовлетворяющее поставленной задаче, а именно: возможность осуществлять взаимодействие удаленных узлов в вычислительной сети, возможность выполнения определенной задачи на конкретном узле, реализация любой схемы взаимодействия удаленных узлов между собой.

					ПГУ 1.090301.29.001 ПЗ			
Изм.	Лист	№ докум.	Подпись	Дата				
Разраб.		Перфильев Д.П.			Программная реализация сети пиринговых узлов для выполнения распределенных приложений Пояснительная записка	Лит.	Лист	Листов
Провер.		Карамышева Н.С.					2	111
Н. Контр		Бычков А.С.				ФВТ, каф ВТ		
Утверд.		Митрохин М.А.				гр. 18BB2		

Содержание

Перечень сокращений и обозначений.....	5
Введение.....	6
1 Программная реализация сети пиринговых узлов для выполнения распределенных приложений	8
1.1 Анализ предметной области	8
1.1.1 Постановка задачи.....	8
1.1.2 Выбор модели распределенной обработки данных.....	8
1.2 Обзор готовых решений	12
1.2.1 Фреймворк JADE.....	13
1.2.2 BOINC	14
1.2.3 LHC Computing Grid	15
1.3 Реализация распределенного приложения	16
1.3.1 Язык программирования C++	16
1.3.2 Библиотека Boost.....	17
1.3.3 Сетевой протокол TCP.....	18
1.4 Алгоритм работы распределенного приложения	19
1.4.1 Конструкции переходов концептуального графа распределенного алгоритма	19
1.4.2 Структура хранения концептуального графа распределенного алгоритма	21
1.5 Архитектура приложения.....	23
1.5.1 Разработка диспетчера узлов	23
1.5.2 Разработка логики работы удаленного узла.....	29
1.5.3 Разработка логики работы оператора	31
1.6 Тестирование и отладка.....	34
1.6.1 Описание работы приложения.....	36
1.6.2 Тестирование работы приложения в локальной сети.....	38
1.6.3 Описание утилиты тестирования	39

1.6.4 Описание результатов тестирования в локальной сети	40
2 Охрана труда.....	43
2.1 Характеристика потенциальных опасностей и вредностей.....	43
2.2 Санитарно-гигиеническая характеристика производства	44
2.3 Электробезопасность	46
2.4 Пожарная безопасность	47
2.5 Расчетно-аналитическая часть	47
Заключение	51
Список использованных источников	52
Приложение А – UML – диаграммы	53
Приложение Б – Листинг программы	58
Приложение В – Презентация.....	102

Перечень сокращений и обозначений

P2P (Peer To Peer) – оверлейная компьютерная сеть, основанная на равноправии участников

TCP (Transmission Control Protocol) – один из основных протоколов передачи данных интернета

UDP (User Datagram Protocol) – один из ключевых элементов набора сетевых протоколов для Интернета

TLS (Transport Layer Security) – криптографический протокол, обеспечивающий защищённую передачу данных между узлами в сети Интернет

UML (Unified Modeling Language) – унифицированный язык моделирования
ГОСТ – региональный стандарт, принятый Межгосударственным советом по стандартизации, метрологии и сертификации Содружества Независимых Государств

ЭВМ –электронно-вычислительная машина

ПК – персональный компьютер

Фреймворк – программная платформа, которая упрощает разработку программного продукта, определяет структуру проекта

					ПГУ 1.090301.29.001 ПЗ	Лист
						5
Изм.	Лист	№ докум.	Подпись	Дата		

Введение

На сегодняшний день обработка и хранение информации имеют особую значимость как в бытовой, так в производственной сферах жизни. Постоянное и постепенное увеличение потребности в обработке колоссальных объемов информации и ограничения вычислительных возможностей аппаратной составляющей компьютеров приводит к необходимости увеличения вычислительной мощности не за счет внутренних ресурсов одного компьютера, а за счет объединения множества вычислительных машин в единую вычислительную сеть.

Пиринговая (*P2P*) сеть – сеть, архитектура которой строится на равноправии участников. Часто в такой сети отсутствуют выделенные серверы, то есть каждый узел выступает в роли сервера и клиента одновременно. Объединение нескольких вычислительных машин позволяет производить параллельные вычисления больших объемов данных, за счет чего уменьшать время затрачиваемое на вычисления и снимать мощностные ограничения для маломощных компьютеров.

Сегодня разработка качественного программного обеспечения невозможна без использования методов параллельного и распределенного программирования.

Понятие «распределенный» изначально относилось к вычислительным сетям, в которых отдельные вычислительные машины были размещены на расстоянии в границах определенной географической области. В настоящее время это понятие несет в себе более широкий смысл, даже применительно к автономным процессам, которые исполняются на одном вычислительном устройстве и контактируют друг с другом с помощью передачи сообщений.

Управление и взаимодействия в распределенных вычислительных сетях в настоящее время обращают на себя внимание огромного числа научных исследователей. Причиной такого ажиотажа к изучению механизмов работы распределенных вычислений является интеграция

метакомпьютерных систем во всевозможных областях, связанных с информационными технологиями, таких как автоматизированные системы управления, нейронные системы распознавания, распределенные сенсорные сети.

Понятие «метакомпьютер» появилось на этапе возникновения высокоскоростных сетевых технологий и подразумевало под собой объединение нескольких неоднотипных вычислительных средств в локальной сети компании для разрешения одной проблемы[1]. В то время главной задачей метакомпьютера являлся поиск наилучшего способа разбиения работы на части и распределения между вычислительными системами с разной архитектурой и мощностью.

В дальнейшем, изучения в области метакомпьютерных технологий изменили направление в сторону однотипного использования вычислительных ресурсов большого количества вычислительных машин в локальной или глобальной сетях. Вычислительными единицами метакомпьютерных систем могут быть как маломощные компьютеры, так и информационные системы, имеющие огромный вычислительный ресурс. Отличительной особенностью такой технологии является факт отсутствия постоянной конфигурации метакомпьютеров, входящих в состав вычислительной системы, то есть отдельные компоненты могут включаться в его конфигурацию или отключаться от нее, при этом технология обеспечивает бесперебойное функционирование системы в целом. Направлением развития современных исследовательских проектов в области метакомпьютерных технологий является обеспечение прозрачного доступа пользователей через Интернет к необходимым распределенным вычислительным ресурсам, а также прозрачного подключения простаивающих вычислительных систем к мета-компьютерам.

					ПГУ 1.090301.29.001 ПЗ	Лист
						7
Изм.	Лист	№ докум.	Подпись	Дата		

1 Программная реализация сети пиринговых узлов для выполнения распределенных приложений

1.1 Анализ предметной области

1.1.1 Постановка задачи

Данная исследовательская работа преследует следующие цели:

- разработка шаблона для лабораторных заданий по дисциплинам, связанным с созданием распределенных сетевых приложений;
- разработка технологии создания распределенных приложений на основании представления работы узла сети, как узла пиринговой системы;
- получение квалифицированных навыков создания сложных сетевых приложений, реализующих произвольный распределенный алгоритм.

К разрабатываемому приложению выставляются следующие требования:

- повышение производительности при решении задачи, посредством распределения вычислений между компьютерами в вычислительной сети;
- удовлетворения требований, представляемых к распределенным вычислительным системам;
- создание алгоритма, устойчивого к исключительным ситуациям, возникающим на узлах сети.

Задачи, которые необходимо решить:

- создание удобного интерфейса пользователя, представленного консольным терминалом;
- реализация алгоритмов передачи данных между узлами;
- разработка алгоритмов, обеспечивающих возможность проведения вычислений на узлах вычислительной сети параллельно.

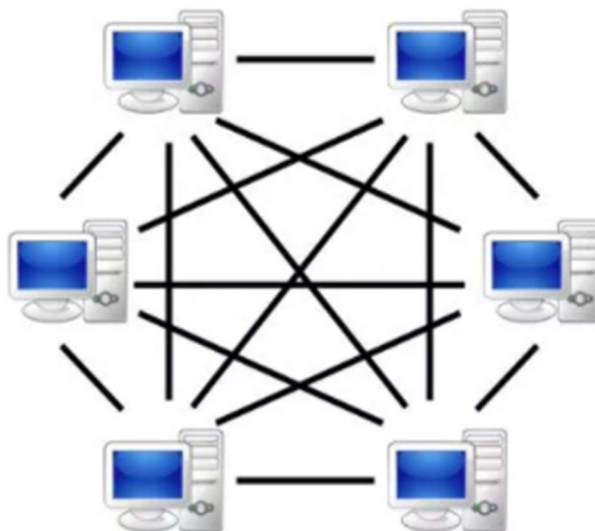
1.1.2 Выбор модели распределенной обработки данных

При разработке распределенных вычислительных систем существует несколько основных используемых технологий.

Клиент-серверная архитектура подразумевает под собой разделение функционала отправки запросов и ответов на них на разных вычислительных машинах в сети, где каждый процесс выполняет свою работу, не завися от других[3]. В данной архитектуре несколько компьютеров, выступающих в роли клиента, посылают запросы и получают ответы на эти запросы от централизованной служебной машины – сервера, которая также может называться хост-системой.

Файл-серверная (ФС) технология – это технология, обеспечивающая работу в сетевом пространстве с доступом к файлам СУБД, хранящимся на сервере. При обработке запроса одного пользователя производится обращение к БД и последующая перекачка данных с блокировкой доступа других пользователей. Затем производится обработка данных на компьютере пользователя.

Пиринговая (*P2P – peer-to-peer networking*) технология – это технология организации одноранговых сетей, часто называемая технологией *P2P*, является одной из самых полезных[2]. В дополнение к модели клиент-сервер, распределенные вычислительные приложения часто используют *P2P*-архитектуру. Пиринговая организация работы группы пользователей основана на понятии «равноправного участника». Появление пиринговых сетей привлекло внимание как простых интернет – пользователей, так и профессионалов сферы компьютерных технологий. В настоящее время больше половины интернет-трафика принадлежит файлообменным сетям *Peet-To-Peer*. Сети *Peer-To-Peer* открывают огромные возможности в сфере информационного обмена. На рисунке 1.1 представлено визуальное представление пиринговой сети.



P2P-network

Рисунок 1.1 – Пиринговая сеть

В данной исследовательской работе выбрана именно пиринговая технология. Главным отличием *P2P* архитектуры от клиент-серверной является наличие логики как клиента, так и сервера на каждом узле, входящим в состав вычислительной сети. На рисунке 1.2 изображено схематическое представление роли узла в пиринговой сети.

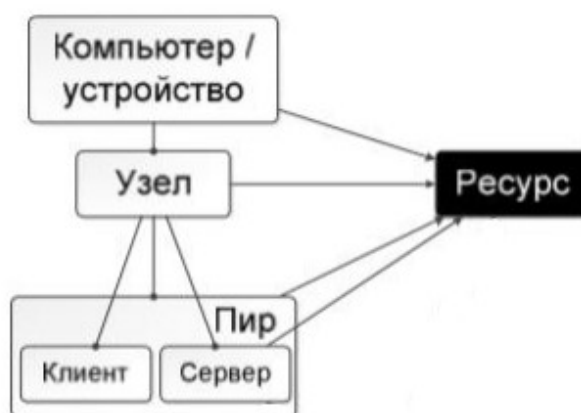


Рисунок 1.2 – Схематическое представление роли узла в сети

Ниже приведены основные недостатки архитектуры «Клиент-Сервер», связанные с присутствием сервера, как центра вычислительной сети, обеспечивающего обработку запросов от множества программ клиентов:

- проблемы масштабируемости. Рост числа клиентов прямо пропорционален с требованиями к мощности вычислительных ресурсов сервера и пропускной способности канала[3]. Данную проблему можно разрешить только с помощью действий направленных на увеличение пропускной способности канала, связывающего клиентов с сервером и улучшения производительности сервера путем использования в его составе более мощных аппаратных ресурсов;

- проблемы зависимости. Бесперебойная и стабильная работа сервера напрямую связана с возможностью функционирования всех клиентов, подключенных к нему. Проблемы, возникшие на сервере отразятся на функциональных возможностях клиентов, что сделает невозможной работу всей вычислительной сети.

Приведенные выше недостатки перекрываются функциональными возможностями *P2P* сетей:

- отсутствие зависимости от централизованных сервисов и ресурсов;
- система может пережить серьезное изменение в структуре сети;
- высокая масштабируемость модели одноранговых вычислений.

Можно выделить следующие основные задачи, которые с легкостью решают *P2P* сети:

- уменьшение и распределение затрат. Наличие в системе одного вычислительного центра подразумевает использование большинства доступных ресурсов (денежных, вычислительных и др.) на поддержание его бесперебойной работы. *P2P* архитектура позволяет перераспределить ресурсы между всеми узлами вычислительной сети. В большинстве случаев узлы являются независимыми, поэтому важно справедливо распределить затраты между ними.

– объединение ресурсов. Каждый узел в системе, использующей *P2P* архитектуру располагает собственными вычислительными ресурсами. Ресурсозатратные программы и приложения требующие большой вычислительной мощности могут использовать объединенные ресурсы всех узлов вычислительной сети для решения своих задач.

– повышенная масштабируемость. Поскольку узлы *P2P* сетей слабо зависят от центрального вычислительного аппарата, появляется возможность увеличения степени масштабируемости сетей и надежности их работы. Объем используемой памяти, количество пользователей, которые могут одновременно пользоваться сетью, количество удаленных узлов, которых можно достичь из текущего узла определяются свойством масштабируемости. Доступность ресурсов, отношение времени работы в к общему времени простоя, количество сбоев в работе сети и так далее, определяются свойством надежности сети. Следовательно, основной задачей становится создание новых алгоритмов обнаружения ресурсов, которые используются в новых *P2P* платформах.

– обеспечение анонимности. Зачастую пользователи сети не хотят, чтобы кто-либо обладал информацией о его нахождении в сети. В архитектурах с использованием сервера, ему необходимо идентифицировать подключившегося клиента, поэтому обеспечение анонимности становится труднодостижимым или невозможным. Противоположностью является *P2P* архитектура, в которой пользователи могут скрыть любую личную информацию.

1.2 Обзор готовых решений

Существует множество решений, позволяющих реализовать алгоритмы распределенных вычислений. Далее будут приведены и кратко разобраны несколько существующих программных продуктов на рынке.

					ПГУ 1.090301.29.001 ПЗ	Лист
						12
Изм.	Лист	№ докум.	Подпись	Дата		

1.2.1 Фреймворк *JADE*

JADE – это программное обеспечение промежуточного слоя, разработанное компанией *TILAB*, предназначенное для создания распределенных мультиагентных приложений на основе транспортной архитектуры «точка-точка». По компьютерам выделенной сети и мобильным устройствам могут быть полностью распределены интеллект, инициатива, информация, ресурсы и контроль. Доступна динамическая работа с узлами, который еще могут называться агентами[4]. Требования программы определяют необходимость появления или исчезновения агентов в системе. Соединения между агентами не зависят от типа сети (проводная, беспроводная). Они являются полностью симметричными, и каждый агент может быть, как отправителем, так и получателем. На рисунке 1.3 изображена роль промежуточного слоя программного обеспечения.

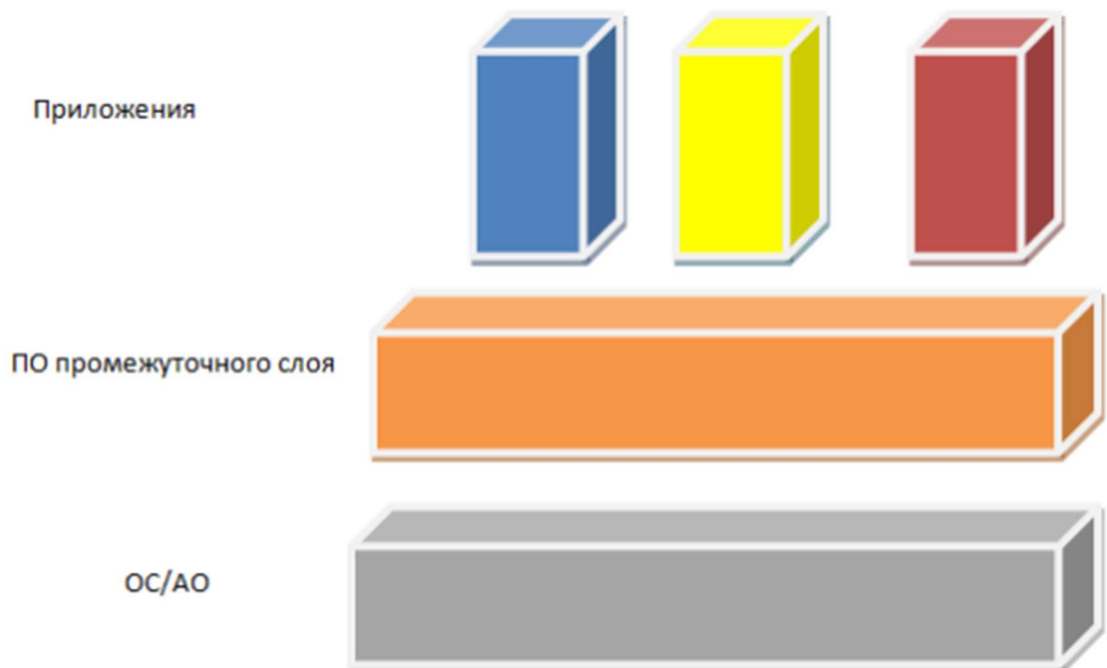


Рисунок 1.3 – Роль промежуточного слоя

JADE представляет базовые службы, используемые в распределенных приложениях типа «точка-точка» в стационарных и мобильных системах. С

помощью платформы *JADE* агенты могут динамически определять друг друга и производить обмен сообщениями. Агенты общаются путем обмена асинхронными сообщениями, модель коммуникаций почти повсеместно применима для распределенных и слабосвязанных коммуникаций, т.е. коммуникаций между гетерогенными сущностями, который ничего не знают друг о друге. Идентификаторами агентов являются их имена и для коммуникации агенты только посылают сообщения друг другу.

В связи с тем, что *JADE* является иностранным программным обеспечением с открытым исходным кодом и текущей политической ситуацией в мире, было принято решение реализовать собственное решение.

1.2.2 Программный комплекс *BOINC*

BOINC — программный комплекс для быстрой организации распределённых вычислений. Состоит из серверной и клиентской частей. Сейчас *BOINC* является доступной платформой для любых проектов, разработанной разработчиками из Калифорнийского университета в Беркли, но первоначально он создавался как часть большого проекта добровольных вычислений — *SETI@home*. Сегодня *BOINC* используется в самых разных областях таких как математика, молекулярная биология, медицина, астрофизика, климатология и т.д. Эта платформа предоставляет ученым доступ к огромным вычислительным мощностям компьютеров со всего мира.

Сервер включает в себя *MySQL*, *HTTP*-сервер с веб-сайтом проекта и некоторый набор программ. *HTTP*-сервер является набором *PHP*-скриптов. Он осуществляет общее управление проектом и выполняет действия, такие как регистрация участников, распределение заданий для обработки, получение результатов, управление базами данных проекта.

BOINC-клиент — универсальный клиент для работы с различными *BOINC*-совместимыми проектами распределённых вычислений. Благодаря одной общей системе управления *BOINC*-клиент может принимать участие сразу в нескольких проектах.

Данное программное обеспечение основано на простой архитектуре типа «клиент-сервер», что влечет за собой и наличие всех недостатков, присущих данной архитектуре при осуществлении распределенных вычислений.

1.2.3 *LHC Computing Grid*

LHC Computing Grid — грид, спроектированный в *CERN* и предназначенный для обработки больших объёмов данных, поступающих с большого адронного коллайдера. В его состав входит 170 вычислительных центров из 36 стран. Грид *LCG* был запущен 3 октября 2008 года, распределённая сеть, которую *CERN* разработал для обработки огромных объёмов данных, получаемых в *LHC*. Она включает в себя как частные волоконно-оптические кабели связи, так и существующие высокоскоростные части общественного Интернета.

Данный проект использует в своей основе одну из форм распределенных вычислений — грид-вычисления. Грид-вычисления — это форма распределённых вычислений, в которой «виртуальный суперкомпьютер» представлен в виде кластеров, соединённых с помощью сети, слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ). Эта технология применяется для решения научных, математических задач, требующих значительных вычислительных ресурсов.

В основе реализации данной формы распределенных вычислений лежит механизм, где грид-клиент получает доступ к удаленному ресурсу (кластеру) или группе компьютеров, который решает задачу[5]. В свою очередь программное обеспечение, разрабатываемое в рамках данной работы использует всю сеть для реализации функции. То есть вся сеть становится единым компьютером. Все узлы связаны и реализуют один алгоритм.

Все рассмотренные выше решения реализуют распределенные вычисления и отвечают требованиям поставленной задачи. Но это готовые,

закрытые решения, скрытые от пользователя, чаще всего коммерческие, в связи с чем было принято решение, реализовать собственный механизм реализации распределенных вычислений.

1.3 Реализация распределенного приложения

При реализации поставленной задачи необходимо обеспечить совместимость с операционной системой *Windows*. Для работы пользователя используется консольный терминал. Необходимо иметь возможность использования многопоточного режима программирования для выполнения параллельных операций на одном узле. Также необходимо обеспечить надежное сетевое соединение между узлами.

На основании выставленных к приложению требований было принято решение использовать язык программирования *C++* с библиотекой *boost*. Обеспечить сетевое соединение на основе сокетов по протоколу *TCP*.

1.3.1 Язык программирования *C++*

C++ – язык программирования общего назначения, созданный датским ученым-программистом Бьёрном Страуструпом как расширение языка программирования *C*, или "*C* с классами". Со временем язык значительно расширился, и современный *C++* теперь обладает объектно-ориентированными, универсальными и функциональными парадигмами в дополнение к средствам для низкоуровневого управления памятью. *C++* был разработан с ориентацией на системное программирование и встроенное программное обеспечение с ограниченными ресурсами и большие системы, требующие производительность, эффективность и гибкость использования в качестве основных моментов его дизайна. *C++* также был признан полезным во многих других контекстах, таких как приложения с ограниченными ресурсами, включая настольные приложения, видеоигры, серверы и приложения, для которых критически важна производительность.

Язык имеет богатую стандартную библиотеку, которая включает в себя распространённые контейнеры и алгоритмы, ввод-вывод, регулярные выражения, поддержку многопоточного режима программирования и другие возможности. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков. C++ является одним из самых популярных языков программирования и широко используется для разработки программного обеспечения.

1.3.2 Библиотека *Boost*

Boost — собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс для лаконичного кодирования различных повседневных подзадач программирования (работа с данными, алгоритмами, файлами, потоками и т. п.)[6]. *Boost* предоставляет бесплатные рецензируемые переносимые библиотеки с исходным кодом C++. *Boost* создают библиотеки, которые хорошо работают со стандартными библиотеками C++. Библиотеки *Boost* предназначены для широкого использования и могут использоваться в широком спектре приложений. Лицензия *Boost* поощряет использование библиотек *Boost* для всех пользователей с минимальными ограничениями. Создатели *Boost* стараются предоставить эталонные реализации, чтобы ее библиотеки были пригодны для последующей стандартизации.

Библиотеки ориентированы на широкий круг пользователей C++ и прикладных областей. Они варьируются от библиотек общего назначения, таких как *smart pointer library* и абстракций операционной системы, таких как *Boost File System* до библиотек, предназначенных в первую очередь для других разработчиков библиотек и продвинутых пользователей C++, таких как *template metaprogramming (MPL)* и *domain-specific language (DSL)*.

1.3.3 Сетевой протокол *TCP*

Протокол управления передачей (*TCP*) является одним из основных протоколов пакета *Internet Protocol Suite*. Он возник в начальной сетевой реализации, в которой он дополнял интернет-протокол *IP*. Поэтому весь пакет обычно называют *TCP/IP*. *TCP* обеспечивает надежную, упорядоченную и проверяемую на ошибки доставку потока октетов (байтов) между приложениями, работающими на хостах, взаимодействующих через *IP*-сеть. Основные интернет-приложения, такие как электронная почта, удаленное администрирование и передача файлов, полагаются на *TCP*, который является частью транспортного уровня из пакета *TCP/IP*. *SSL/TLS* часто работает поверх *TCP*.

TCP ориентирован на соединение, и оно между клиентом и сервером устанавливается перед отправкой данных. Сервер должен прослушивать запросы на соединение от клиентов, прежде чем соединение будет установлено[7]. Трехстороннее рукопожатие, повторная передача и обнаружение ошибок повышают надежность, но увеличивают задержку. Приложения, которым не требуется надежная служба потока данных, могут использовать протокол пользовательских дейтаграмм (*UDP*), который обеспечивает службу дейтаграмм без подключения, которая отдает приоритет времени над надежностью. *TCP* использует предотвращение перегрузки сети. Тем не менее, существуют уязвимости *TCP*, включая отказ в обслуживании или захват соединения.

TCP очень хорошо подходит для передачи больших файлов. Две конечные точки открывают соединение, данные файла записываются одним концом в соединение и считываются другим концом, а функции выше гарантируют, что файл будет получен правильно. *TCP* также хорошо работает для интерактивных приложений, где каждая сторона отправляет и получает потоки небольших пакетов. Примеры этого включают *ssh* или *telnet*, где пакеты обмениваются при каждом нажатии клавиши, и соединения с

базой данных, которые могут передавать много запросов в секунду. *TCP* даже работает достаточно хорошо для запроса или ответа протоколы, когда одна сторона отправляет сообщение, другая сторона отвечает, и соединение закрывается. Однако недостатком здесь являются накладные расходы на настройку нового соединения для каждого запроса; лучшим решением прикладного протокола может быть разрешение нескольких пар запрос или ответ по одному *TCP*-соединению.

1.4 Алгоритм работы распределенного приложения

1.4.1 Конструкции переходов концептуального графа распределенного алгоритма

Программа реализует взаимодействие узлов по схеме на основе концептуального графа. Граф может быть построен с помощью следующих конструкций переходов:

- линейный переход – прямая передача управления и данных от одного оператора к другому;
- условный переход – передача данных и управления одному из нескольких операторов в зависимости от результата вычислений на предыдущем операторе;
- параллельный переход – одновременная передача управления и данных сразу нескольким операторам;
- *Switch-Or*. Фрагмент «*Switch-Or*» реализует передачу сообщения по одной из ветвей распределенного алгоритма по значению переменной k [8].

Ниже изображены примеры графов с линейным переходом, условным переходом, параллельным переходом, переходом «*Switch-Or*» на рисунках 1.4, 1.5, 1.6, 1.7 соответственно.

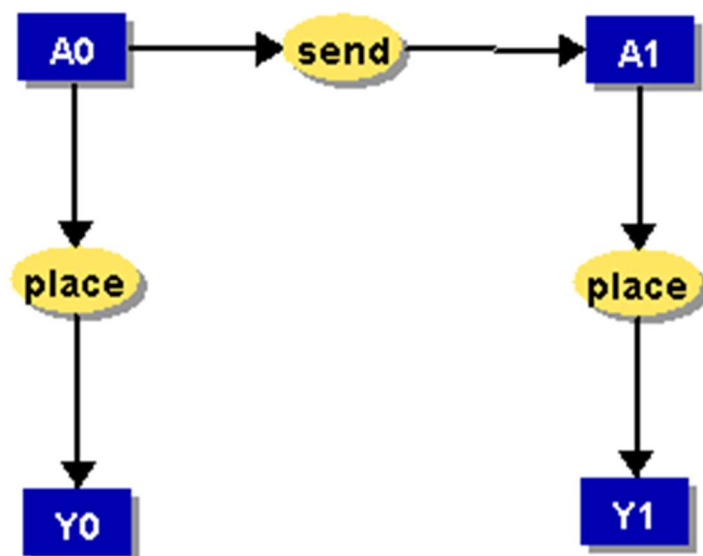


Рисунок 1.4 - Линейный переход

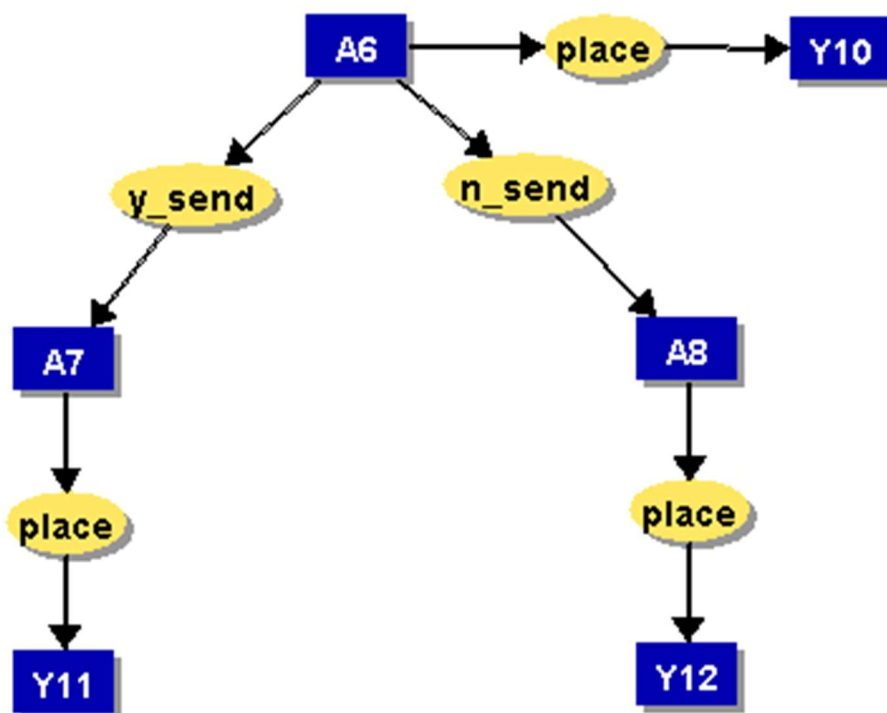


Рисунок 1.5 - Условный переход

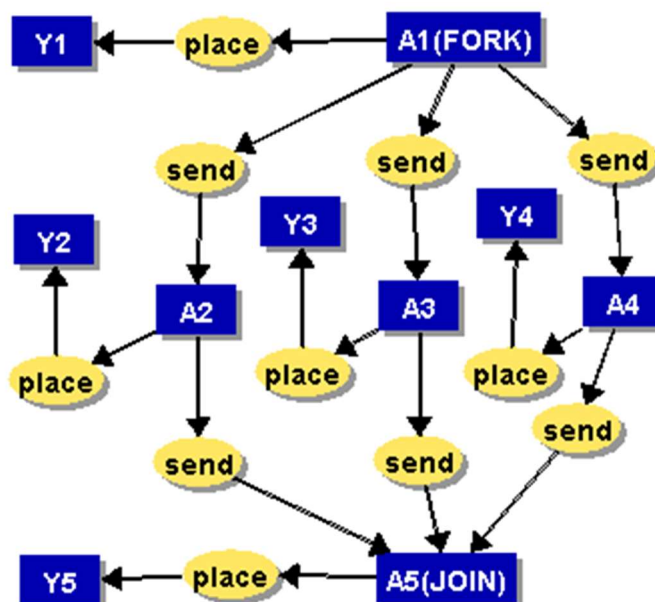


Рисунок 1.6 - Параллельный переход

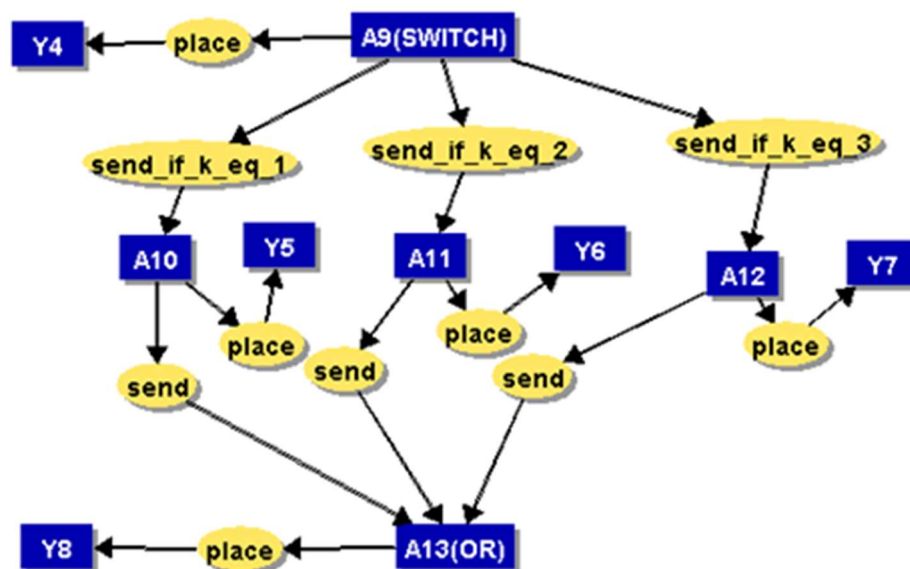


Рисунок 1.7 – Переход «Switch-Or»

1.4.2 Структура хранения концептуального графа распределенного алгоритма

Для хранения текстового представления концептуального графа выбран текстовый файл. Программа способна реализовывать любые схемы

взаимодействия узлов сети. Для изменения схемы взаимодействия достаточно изменить конфигурационный файл.

Данные записываются в файл по следующей схеме:

[Оператор] ; [Узел] ; [Родительские операторы] ; [Дочерние операторы]

В случае отсутствия родительских или дочерних операторов допускается ничего не указывать на месте соответствующих полей, тогда формат записи данных может иметь следующий вид:

A0;Y0;;

Пропуск полей оператора и узла не допускается.

Описание схематических обозначений приведены в таблице 1.1.

Таблица 1.1 – Описание обозначений

Схематическое обозначение	Описание
[Оператор]	Порядковый номер оператора
[Узел]	Номер узла, на котором располагается оператор
[Родительские операторы]	Операторы, от которых текущий оператор ожидает передачи управления
[Дочерние операторы]	Операторы, которым передает управление текущий оператор

Родительских и дочерних узлов может быть несколько, и для обозначения различных конструкций переходов в графе используются следующие обозначения:

- слеш « / » - для обозначения параллельного перехода;
- запятая « , » - для обозначения условного перехода.

Для графа, приведенного на рисунке 1.6, конфигурационный файл будет выглядеть следующим образом:

A1;Y1;;A2/A3/A4

A2;Y2;A1;A5

A3;Y3;A1;A5

A4;Y4;A1;A5

A5;Y5;A2/A3/A4;

1.5 Архитектура приложения

В данной работе приведены результаты разработки программной реализации сети пиринговых узлов для выполнения распределенных вычислений.

Разработку программного обеспечения можно разделить на следующие этапы:

- разработка диспетчера узлов;
- разработка логики работы узла;
- разработка логики работы оператора.

1.5.1 Разработка диспетчера узлов

Диспетчер узлов должен выполнять следующие задачи:

- загружать задачу для пиринговой сети из файла конфигурации;
- принимать подключения от доступных узлов сети;
- рассылать задачи подключенным узлам;
- принимать подключения от операторов на узлах;
- обрабатывать запросы от операторов на получение адресов других операторов;
- отслеживать статус работы операторов.

На рисунке 1.8 представлена *UML*-диаграмма вариантов использования программы диспетчера узлов.



Рисунок 1.8 - Диаграмма вариантов использования программы
диспетчера узлов

Схема концептуального графа заносится в текстовый файл и находится на сервере. На сервере реализован класс «*TaskParser*», который осуществляет загрузку и разбор файла конфигурации. Ниже приведена реализация класса «*TaskParser*».

```

class TaskParser
{
public:
    TaskParser( const std::string& configFilepath );
    std::vector<Task> GetTasks();
private:
    bool ParseString( const std::string& strTask,
AgentData& agent, std::string& nodeName );
private:
    std::ifstream configFile_;
};
  
```

Метод «*TaskParser::ParseString*» разбирает строку текстового файла и заполняет данными структуру «*AgentData*». Реализация структуры «*AgentData*» будет разобрана далее. Метод «*TaskParser::GetTasks*» возвращает список задач в виде вектора структур «*Task*». Ниже приведена реализация структуры «*Task*»:

```
struct Task
{
    std::string nodeName;
    int numAgents = 0;
    std::vector<AgentData> agents;
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize( Archive& ar, const unsigned version )
    {
        ar& nodeName& numAgents& agents;
    }
};
```

Для сетевого взаимодействия с узлами в программе диспетчера узлов реализован класс «*ServerSocket*». В данном классе реализованы методы для выполнения основных задач сетевого взаимодействия, таких как прослушивание подключений, отправка и прием сообщений, корректного закрытия соединения. Ниже приведена реализация класса «*ServerSocket*»:

```
class ServerSocket
{
public:
    void ServerInit( int port );
    tcp::socket Listen();
    static void CloseClientSockets(
std::vector<tcp::socket>& socks );
    static void BroadcastMessage( const std::string& msg,
std::vector<tcp::socket>& socks );
    static int Send( const char* buf, int size,
tcp::socket& client );
```

```

        static std::string Recieve( tcp::socket& client );
        ~ServerSocket();
private:
        asio::io_service ioService_;
        tcp::acceptor* acceptor_ = 0;
};

```

Класс «*ServerSocket*» включает в себя методы для установления и поддержания соединения с удаленным узлом. Ниже приводится краткое описание работы каждого метода.

«*ServerInit*» - метод для первоначальной инициализации диспетчера узлов.

«*Listen*» - метод, осуществляющий запуск прослушивания новых подключений удаленных узлов.

«*CloseClientSockets*» - метод, осуществляющий закрытие соединения с удаленными узлами.

«*BroadcastMessage*» - метод, осуществляющий рассылку всем подключенным удаленным узлам сигнала о завершении процесса распределенных вычислений успешно или с ошибкой.

«*Send*» - метода для отправки данных.

«*Recieve*» - метод для приема данных.

Основная логика взаимодействия с удаленным узлом заключается в отправке ему задачи и реализована в функции «*InitNode*». Ниже приводится реализация функции «*InitNode*»:

```

void InitNode( tcp::socket* socket, Task task )
{
    char* bytes = nullptr;
    try
    {
        std::stringstream ss = Serialization( task );
        ss.seekg( 0, std::ios::end );
        int size = ss.tellg();
    }
}

```

```

        ss.seekg( 0, std::ios::beg );
        bytes = new char[ size ];
        memcpy( bytes, ss.str().c_str(), size );
        ServerSocket::Send( bytes, size, *socket );
        output_locker.lock();
        std::cout << "Задача успешно выслана узлу: " <<
OutputSocketAddress( *socket ) << std::endl;
        output_locker.unlock();
        std::string      addr      =      socket->
remote_endpoint().address().to_string();
        ServerSocket::Send( addr.c_str(), addr.size() + 1,
*socket );

        for( int i = 0; i < task.numAgents; i++ )
        {
            std::string address = ServerSocket::Recieve(
*socket );

            std::tuple<std::string, std::string, int>
addr = ParseAddress( address );
            agentAddressTable.SetAddress( addr );
        }
        delete[] bytes;
    }
    catch( std::exception& ex )
    {
        std::string error = "Ошибка инициализации узла: "
+ task.nodeName + "\n" + std::string( ex.what() );
        delete[] bytes;
        throw std::exception( error.c_str() );
    }
}

```

Взаимодействие диспетчера агентов с конкретным оператором реализована в функции «*AgentHandler*» и заключается в обработке запросов от оператора. Диспетчер узлов может обработать сигнал о завершении

вычислений успешно или с ошибкой, а также запрос на получение сетевого адреса оператора. Реализация функции «*AgentHandler*» приведена ниже:

```
void AgentHandler( tcp::socket* agentSock, bool
isStartThread )
{
    try
    {
        if( isStartThread )
        {
            Sleep(500);
            std::cout << "Нажмите любую клавишу для
запуска процесса вычислений..." << std::endl;
            std::getchar();
            ServerSocket::Send( "start", 6, *agentSock );
        }
        while( true )
        {
            std::string flag;
            std::string op;
            size_t pos;
            std::string request = ServerSocket::Recieve(
*agentSock );

            if( request == "finish" )
            {
                isFinish = true;
                continue;
            }
            if( ( pos = request.find( ';' ) ) !=
std::string::npos )
            {
                flag = request.substr( 0, pos );
                op = request.substr( pos + 1 );
            }
            if( flag == "error" )
```

```

        {
            std::string error = "Агент " + op + "
завершился с ошибкой\n";
            throw std::exception( error.c_str() );
        }
        else if( flag == "success" )
        {
            output_locker.lock();
            std::cout << "Агент " << op << "
завершился успешно\n";
            output_locker.unlock();
            break;
        }
        else
        {
            std::string addr =
agentAddressTable.GetAddressByName( request );
            ServerSocket::Send( addr.c_str(),
addr.size() + 1, *agentSock );
        }
    }
}
catch( std::exception& ex )
{
    throw std::exception( ex.what() );
}
}

```

1.5.2 Разработка логики работы удаленного узла

Логика узла выполняется в основном потоке приложения. Основной задачей разработки программной логики узла является соединения с диспетчером узлов и получения от него задачи. На рисунке 1.9 представлена *UML*-диаграмма вариантов использования программы удаленного узла.



Рисунок 1.9 - диаграмма вариантов использования программы удаленного узла

В задаче указывается количество операторов, выполняемых на текущем узле. Когда узлу станет известна его задача, программа запустит требуемое количество операторов в отдельных потоках.

Реализация сетевого соединения с диспетчером узлов находится в классе «*ClientSocket*». Реализация класса приводится ниже.

```

class ClientSocket
{
public:
    ClientSocket()
    {
        clientSocket_ = new tcp::socket( ioService_ );
    };
    int Send( const char* buf, int size );
    std::string Recieve();
    void Connect( const std::string& ip, int port );
    void Disconnect();
    ~ClientSocket();
}
  
```

```
private:
    tcp::endpoint endpoint_;
    tcp::socket* clientSocket_ = 0;
    asio::io_service ioService_;
};
```

При создании нового объекта класса «*ClientSocket*» создается новый объект сокета, который хранится как приватная переменная этого класса. Таким образом один экземпляр класса «*ClientSocket*» представляет отдельную сетевую единицу, и включает в себя методы для работы с сокетом. Методы класса схожи по названию и предназначению с методами класса «*ServerSocket*», рассмотренного выше, за исключением методов, рассмотренных ниже.

«*Connect*» - метод, осуществляющий подключение к серверу по необходимому сетевому адресу.

«*Disconnect*» - метод для корректного отсоединения от сервера.

1.5.3 Разработка логики работы оператора

Основной задаче оператора является прием управления от родительских операторов, обработка данных и дальнейшая передача управления дочерним операторам. На рисунке 1.9 представлен порядок действий оператора удаленного узла.



Рисунок 1.9 – Схема программы оператора удаленного узла

Весь программный модуль работы оператора заключен в классе «*Agent*». В нем реализован основной функционал, необходимый для функционирования оператора. Ниже приведена реализация класса «*Agent*»:

```
class Agent
{
public:
    Agent( AgentData data )
        : data_( data )
    {};
    Agent(Agent&& ) = default;

    void StartAgent( std::function<void( const std::string&
)> task, bool isFirstAgent );
    void StopAgent();
    ~Agent() { StopAgent(); };

private:
    void FirstAgentThread( std::function<void( const
std::string& )> task );
    void AgentThread( std::function<void( const
std::string& )> task );
    std::vector<std::string> WaitParents( ServerSocket&
socket, int port, int parentsCount );
    void CheckParentsAccuracy( std::vector<std::string>&
recievedParents );
    void DelegateChilds( ClientSocket& socket,
std::vector<std::pair<std::string, int>>& childs, const
std::string& data );
private:
    ClientSocket* clientSock_ = 0;
    ServerSocket* serverSock_ = 0;
    AgentData data_;
};
```

Ниже приводится краткое описание основных членов класса:

- «*AgentThread*» - основной метода класса, описывает основную логику работы оператора;
- «*WaitParents*» - метод класса, осуществляющий ожидания управления от родительских операторов;
- «*DelegateChilds*» - метод класса, выполняющий передачу управления дочерним операторам.

При создании объекта класса, ему передается структура «*AgentData*». В ней содержится информация о положении текущего оператора в схеме распределенной вычислительной сети, информация о соседних операторах и номер порта, который будет прослушивать сокет текущего оператора. Ниже приводится программная реализация структуры «*AgentData*».

```
struct AgentData
{
    std::string agentName;
    int listenPort = 0;
    bool isChildsFork = false;
    bool isParentsFork = false;
    std::vector<std::string> parentAgents;
    std::vector<std::string> childAgents;
    friend class boost::serialization::access;
    template<typename Archive>
    void serialize( Archive& ar, const unsigned version )
    {
        ar& agentName& listenPort& isChildsFork&
isParentsFork& parentAgents& childAgents;
    }
};
```

В таблице 1.2 приведено краткое описание полей структуры.

					ПГУ 1.090301.29.001 ПЗ	Лист
						33
Изм.	Лист	№ докум.	Подпись	Дата		

Таблица 1.2 – Описание полей структуры «AgentData»

Название переменной	Описание
«agentName»	Имя агента
«listenPort»	Номер порта, который прослушивает сокет текущего агента
«isChildFork»	Логическая переменная, сигнализирующая является ли переход к дочерним узлам параллельным
«isParentFork»	Логическая переменная, сигнализирующая является ли переход от родительских операторов параллельным
«parentAgents»	Вектор, содержащий имена родительских операторов
«childAgents»	Вектор, содержащий имена дочерних операторов

1.6 Тестирование и отладка

Данный раздел содержит описание тестирования работы приложения. Работа оператора при тестировании эмитируется и представляет собой вывод данных об узле на экран консоли. Было проведено тестирование на виртуальной сети на одном компьютере без учета времени работы сети, а также тестирование на реальной локальной сети с учетом времени работы сети. Тестирование проводилось в одной из аудиторий кафедры

«Вычислительная техника» Пензенского государственного университета. Для проведения тестирования был использован граф приведенный на рисунке 1.11.

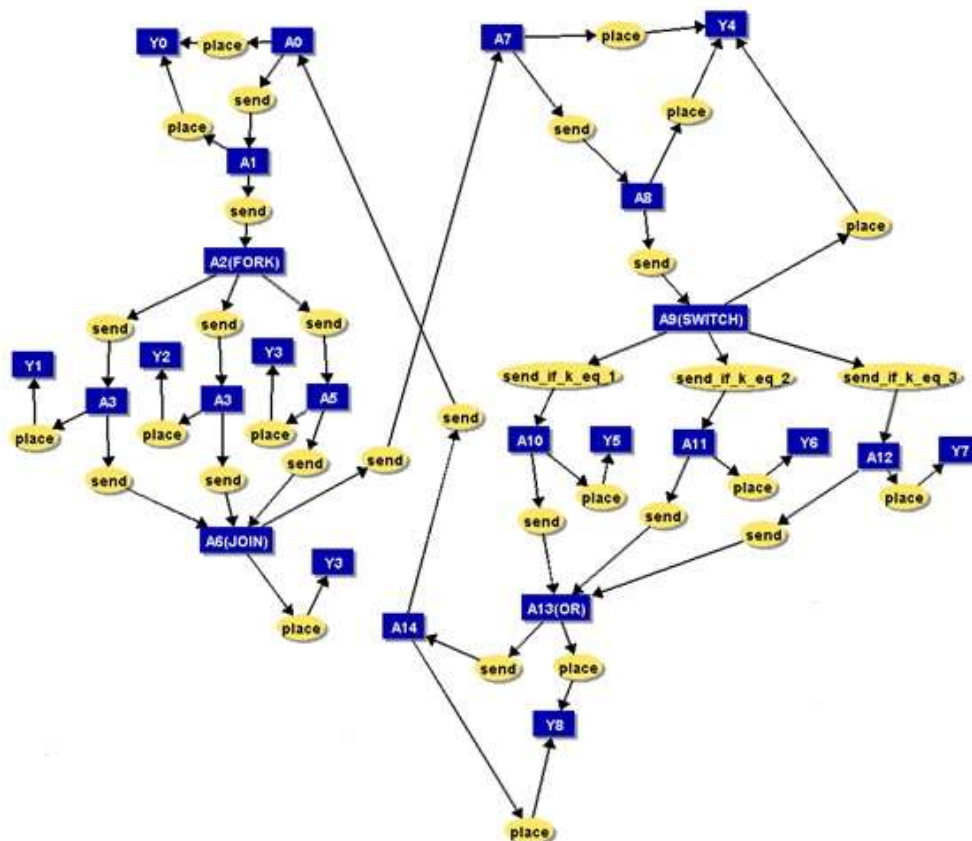


Рисунок 1.11 - Концептуальный граф, используемый для тестирования приложения

По схеме соединения операторов в графе, приведенном на рисунке 1.11, был создан конфигурационный файл приложения, содержание которого приведено ниже.

A0;Y0;A14;A1
 A1;Y0;A0;A2
 A2;Y0;A1;A3/A4/A5
 A3;Y1;A2;A6
 A4;Y2;A2;A6

A5;Y3;A2;A6

A6;Y3;A3/A4/A5;A7

A7;Y4;A6;A8

A8;Y4;A7;A9

A9;Y4;A8;A10,A11,A12

A10;Y5;A9;A13

A11;Y6;A9;A13

A12;Y7;A9;A13

A13;Y8;A10,A11,A12;A14

A14;Y8;A13;A0

1.6.1 Описание работы приложения

Для описания работы был произведен запуск на одном компьютере программы диспетчера узлов и программ самих узлов. На рисунке 1.12 приведен вывод консоли программы диспетчера узлов.

По рисунку программы можно понять порядок действий программы диспетчера узлов. После запуска диспетчера узлов, он разбирает конфигурационный файл и формирует структуру задач для каждого оператора. Далее диспетчер начинает ожидать подключения узлов. После того как все узлы подключены начинается рассылка задач узлам.

Далее диспетчер узлов начинает ожидать подключения агентов (операторов). После того как все агенты подключились диспетчер узлов переходит в режим ожидания. Он ожидает сигнала от каждого агента об успешном выполнении или появления ошибки. После получения сигналов от всех агентов диспетчер узлов рассылает всем узлам сигнал об успешном или аварийном завершении. На этом вычисления завершаются.

```

C:\WINDOWS\system32\cmd.exe
Сервер запущен
Конфигурационный файл успешно разобран
Прослушивание подключений...
Новый узел подключен - 127.0.0.1:1059 - Y0
Новый узел подключен - 127.0.0.1:1061 - Y1
Новый узел подключен - 127.0.0.1:1062 - Y2
Новый узел подключен - 127.0.0.1:1063 - Y3
Новый узел подключен - 127.0.0.1:1064 - Y4
Новый узел подключен - 127.0.0.1:1065 - Y5
Новый узел подключен - 127.0.0.1:1066 - Y6
Новый узел подключен - 127.0.0.1:1067 - Y7
Новый узел подключен - 127.0.0.1:1068 - Y8
Требуемое число клиентов подключено

Рассылка задач узлам...
Задача успешно выслана узлу: 127.0.0.1:1065
Задача успешно выслана узлу: 127.0.0.1:1062
Задача успешно выслана узлу: 127.0.0.1:1066
Задача успешно выслана узлу: 127.0.0.1:1061
Задача успешно выслана узлу: 127.0.0.1:1063
Задача успешно выслана узлу: 127.0.0.1:1059
Задача успешно выслана узлу: 127.0.0.1:1067
Задача успешно выслана узлу: 127.0.0.1:1068
Задача успешно выслана узлу: 127.0.0.1:1064
Все задачи успешно разосланы узлам

Работа агентов...
Новый агент подключен - 127.0.0.1:1069 - A0
Новый агент подключен - 127.0.0.1:1073 - A1
Новый агент подключен - 127.0.0.1:1071 - A2
Новый агент подключен - 127.0.0.1:1074 - A3
Новый агент подключен - 127.0.0.1:1072 - A4
Новый агент подключен - 127.0.0.1:1075 - A5
Новый агент подключен - 127.0.0.1:1078 - A6
Новый агент подключен - 127.0.0.1:1076 - A7
Новый агент подключен - 127.0.0.1:1081 - A8
Новый агент подключен - 127.0.0.1:1082 - A9
Новый агент подключен - 127.0.0.1:1084 - A10
Новый агент подключен - 127.0.0.1:1080 - A11
Новый агент подключен - 127.0.0.1:1085 - A12
Новый агент подключен - 127.0.0.1:1079 - A13
Новый агент подключен - 127.0.0.1:1086 - A14
Агент A1 завершился успешно
Агент A2 завершился успешно
Агент A3 завершился успешно
Агент A4 завершился успешно
Агент A5 завершился успешно
Агент A6 завершился успешно
Агент A7 завершился успешно
Агент A8 завершился успешно
Агент A9 завершился успешно
Агент A12 завершился успешно
Агент A13 завершился успешно
Агент A14 завершился успешно
Агент A0 завершился успешно
Работа агентов успешно завершена

```

Рисунок 1.12 - Вывод консоли программы диспетчера узлов

Вывод консоли удаленного узла также содержит подробные сведения о своей работе и приведен на рисунке 1.13.

```

Соединение с диспетчером узлов установлено
Ожидание задачи от диспетчера узлов
Задача получена
Узел: Y0
Операторы узла: A0 A1 A2

[ A0 ] Ожидание сигнала о начале вычислений от диспетчер узлов
[ A1 ] Ожидание данных от родительских узлов: A0
[ A2 ] Ожидание данных от родительских узлов: A1
[ A0 ] Сигнал старта получен
Оператор A0 успешно выполнен
[ A0 ] Отправка данных дочерним узлам
[ A0 ] Данные успешно отправлены дочерним узлам: A1
[ A0 ] Ожидание данных от родительских узлов: A14
[ A1 ] Данные получены: A0
Оператор A1 успешно выполнен
[ A1 ] Отправка данных дочерним узлам
[ A1 ] Данные успешно отправлены дочерним узлам: A2
[ A2 ] Данные получены: A1
Оператор A2 успешно выполнен
[ A2 ] Отправка данных дочерним узлам
[ A2 ] Данные успешно отправлены дочерним узлам: A3 A4 A5
[ A0 ] Данные получены: A14
Вычисления завершены успешно.

```

Рисунок 1.13 - Вывод консоли программы удаленного узла

На рисунке 1.13 приведен вывод консоли удаленного узла на примере узла Y0. Порядок действий остальных узлов является аналогичным. Первоначально происходит попытка установления соединения с диспетчером узлов. При успешном соединении узел переходит в режим ожидания задачи. Как только задача будет получена узел запустит отдельные потоки для выполнения своих агентов (операторов), а сам перейдет в режим ожидания сигнала завершения от диспетчера узлов.

Жизненный цикл агента состоит из ожидания данных от родительских агентов, выполнения функционала самого оператора и передачи данных дочерним агентам. После выполнения агент сообщает диспетчеру узлов о своем успешном завершении или о возникновении ошибки.

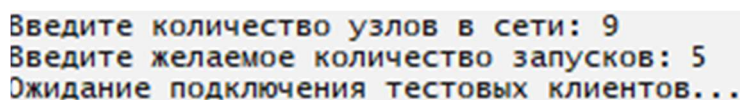
1.6.2 Тестирование работы приложения в локальной сети

Для проведения тестирования и определения среднего времени работы приложения в реальной локальной сети была разработана утилита тестирования. Далее приводится подробное описание данной утилиты и результатов тестирования.

1.6.3 Описание утилиты тестирования

Приложение для тестирования вычислительной сети состоит из программы сервера и программы клиента. Основной задачей приложения для тестирования является запуск программы диспетчера узлов и программ удаленных узлов требуемое количество раз.

Для проведения тестов с использованием данной утилиты необходимо запустить программу сервер на одном из компьютеров. После запуска сервера для тестирования необходимо ввести количество удаленных узлов, используемых в работе распределенной сети и желаемое количество запусков. Далее сервер начинает ожидать подключения программ клиентов. На рисунке 1.14 показан вывод консоли программы сервера для тестирования.



```
Введите количество узлов в сети: 9
Введите желаемое количество запусков: 5
Ожидание подключения тестовых клиентов...
```

Рисунок 1.14 - Вывод консоли сервера для тестирования

Далее необходимо запустить на каждом узле программу клиента для тестирования. После запуска требуемого числа клиентов и их успешного подключения сервер запустит диспетчер узлов и отправит сигнал клиентам, получив который, клиенты запустят программы удаленных узлов на своих устройствах. Далее выполниться логика работы распределенного приложения, описанная выше. После завершения работы диспетчера узлов сервер для тестирования снова запустит диспетчер узлов и также разошлет сигналы запуска подключенным клиентам. Такой алгоритм повторится требуемое количество раз. На рисунке 1.15 показан вывод консоли программы клиента для тестирования.

```
C:\Users\Иван\Desktop\УЧЕБА\4 курс\Диплом\test drive\TestClient.exe
Клиент успешно подключен
Запуск программы узла 1
Запуск программы узла 2
Запуск программы узла 3
Запуск программы узла 4
Запуск программы узла 5
Корректное завершение тестов
```

Рисунок 1.15 - Вывод консоли клиента для тестирования

Для успешной работы утилиты для тестирования исполняемый файл диспетчера узлов и конфигурационный файл должны находиться в одном каталоге вместе с исполняемым файлом программы сервера для тестирования. В свою очередь исполняемый файл программы удаленного узла должен находиться в одном каталоге вместе исполняемым файлом программы клиента для тестирования.

1.6.4 Описание результатов тестирования в локальной сети

Для тестирования в реальных условиях было задействовано 10 компьютеров, объединенных в локальную сеть. Один из компьютеров являлся главным и осуществлял функцию управления. На него были помещены исполняемые файлы диспетчера узлов и программы сервера для тестирования и конфигурационный файл, содержащий текстовое представление графа. Остальные 9 вычислительных машин выступали в роли удаленных узлов и на них были помещены исполняемые файлы программы удаленных узлов и программы клиента для тестирования.

Ручной единичный запуск приложения показал успешную работу приложения с учетом реальной сетевой нагрузки на приложение. Вывод консоли диспетчера узлов при работе в локальной сети приведен на рисунке 1.16.

```

Сервер запущен
Конфигурационный файл успешно разобран
Прослушивание подключений...
Новый узел подключен - 192.168.10.49:7609 - Y0
Новый узел подключен - 192.168.10.30:1300 - Y1
Новый узел подключен - 192.168.10.50:2191 - Y2
Новый узел подключен - 192.168.10.32:1286 - Y3
Новый узел подключен - 192.168.10.42:2153 - Y4
Новый узел подключен - 192.168.10.43:29740 - Y5
Новый узел подключен - 192.168.10.44:1398 - Y6
Новый узел подключен - 192.168.10.45:1789 - Y7
Новый узел подключен - 192.168.10.46:1434 - Y8
Требуемое число клиентов подключено

Рассылка задач узлам...
Задача успешно выслана узлу: 192.168.10.49:7609
Задача успешно выслана узлу: 192.168.10.30:1300
Задача успешно выслана узлу: 192.168.10.50:2191
Задача успешно выслана узлу: 192.168.10.32:1286
Задача успешно выслана узлу: 192.168.10.42:2153
Задача успешно выслана узлу: 192.168.10.43:29740
Задача успешно выслана узлу: 192.168.10.44:1398
Задача успешно выслана узлу: 192.168.10.45:1789
Задача успешно выслана узлу: 192.168.10.46:1434
Все задачи успешно разосланы узлам

Работа агентов...
Новый агент подключен - 192.168.10.49:7610 - A0
Новый агент подключен - 192.168.10.50:2192 - A1
Новый агент подключен - 192.168.10.49:7611 - A2
Новый агент подключен - 192.168.10.49:7612 - A3
Новый агент подключен - 192.168.10.43:29741 - A4
Новый агент подключен - 192.168.10.42:2154 - A5
Новый агент подключен - 192.168.10.44:1399 - A6
Новый агент подключен - 192.168.10.45:1790 - A7
Новый агент подключен - 192.168.10.42:2155 - A8
Новый агент подключен - 192.168.10.42:2156 - A9
Новый агент подключен - 192.168.10.46:1435 - A10
Новый агент подключен - 192.168.10.46:1436 - A11
Новый агент подключен - 192.168.10.30:1301 - A12
Новый агент подключен - 192.168.10.32:1288 - A13
Новый агент подключен - 192.168.10.32:1287 - A14
Агент A1 завершился успешно
Агент A2 завершился успешно
Агент A4 завершился успешно
Агент A3 завершился успешно
Агент A5 завершился успешно
Агент A6 завершился успешно
Агент A7 завершился успешно
Агент A8 завершился успешно
Агент A9 завершился успешно
Агент A12 завершился успешно
Агент A13 завершился успешно
Агент A14 завершился успешно
Агент A0 завершился успешно
Работа агентов успешно завершена

```

Рисунок 1.16 - Вывод консоли диспетчера узлов при работе в локальной сети

Далее было проведено 100 запусков приложения. Было получено время работы сети при каждом запуске и вычислено среднее время работы

приложения с учетом реальных задержек сети. На рисунке 1.17 показан вывод консоли программы сервера для тестирования.

```

Время работы сети при 60 запуске - 3.608682 секунд
Время работы сети при 61 запуске - 3.608135 секунд
Время работы сети при 62 запуске - 3.769100 секунд
Время работы сети при 63 запуске - 3.057100 секунд
Время работы сети при 64 запуске - 3.071273 секунд
Время работы сети при 65 запуске - 3.284292 секунд
Время работы сети при 66 запуске - 2.713378 секунд
Время работы сети при 67 запуске - 3.528337 секунд
Время работы сети при 68 запуске - 2.810711 секунд
Время работы сети при 69 запуске - 3.593142 секунд
Время работы сети при 70 запуске - 3.834504 секунд
Время работы сети при 71 запуске - 2.948805 секунд
Время работы сети при 72 запуске - 2.904773 секунд
Время работы сети при 73 запуске - 2.518278 секунд
Время работы сети при 74 запуске - 2.793271 секунд
Время работы сети при 75 запуске - 3.724942 секунд
Время работы сети при 76 запуске - 3.645432 секунд
Время работы сети при 77 запуске - 3.789274 секунд
Время работы сети при 78 запуске - 3.551512 секунд
Время работы сети при 79 запуске - 3.833070 секунд
Время работы сети при 80 запуске - 2.591254 секунд
Время работы сети при 81 запуске - 3.571357 секунд
Время работы сети при 82 запуске - 3.302580 секунд
Время работы сети при 83 запуске - 3.489278 секунд
Время работы сети при 84 запуске - 3.438564 секунд
Время работы сети при 85 запуске - 3.417757 секунд
Время работы сети при 86 запуске - 3.029633 секунд
Время работы сети при 87 запуске - 3.258103 секунд
Время работы сети при 88 запуске - 2.780892 секунд
Время работы сети при 89 запуске - 3.211951 секунд
Время работы сети при 90 запуске - 3.045681 секунд
Время работы сети при 91 запуске - 2.976229 секунд
Время работы сети при 92 запуске - 3.300312 секунд
Время работы сети при 93 запуске - 2.981066 секунд
Время работы сети при 94 запуске - 3.802705 секунд
Время работы сети при 95 запуске - 2.708492 секунд
Время работы сети при 96 запуске - 3.033126 секунд
Время работы сети при 97 запуске - 3.022849 секунд
Время работы сети при 98 запуске - 3.481389 секунд
Время работы сети при 99 запуске - 3.024569 секунд
Среднее время работы сети при 100 запусках - 3.227716 секунд
Корректное завершение тестов

```

Рисунок 1.17 - Вывод консоли программы сервера для тестирования

2 Охрана труда

Реализация любого проекта, будь то полноценный производственный процесс или работа над конкретной задачей, требует не только экономических затрат и умственных или физических усилий, но и обеспечения труда в безопасных для здоровья работников условиях. Для того, чтобы предоставить равные условия для работы, была разработана система правил и мер, которая легла в основу целой науки, основной целью которой стали систематизация знаний и обучение квалифицированных работников охране труда. Безопасность жизнедеятельности представляет собой свод правил, важнейшей целью которых является разработка и обновление мер по предупреждению производственного травматизма и заболеваемости.

В случае работы на ЭВМ, основными факторами, влияющими на работников, будут особенности организации рабочего пространства, удобство локации, а также самого места работы, условиями внутреннего расположения человека в помещении (освещение рабочего места, уровень шума, качеством техники и др.), общий характер работы.

В данной главе будут рассмотрены опасности, которые могут возникнуть в процессе реализации данного проекта, пожарные и санитарные нормы, применяемые к помещению, где должен реализовываться проект, санитарно-гигиеническая характеристика места разработки проекта, а также основные требования электробезопасности.

2.1 Характеристика потенциальных опасностей и вредностей

Для того, чтобы определить основные потенциальные опасности, присутствующие при разработке данного проекта, нужно определить, в каких условиях может производиться данный проект. Данные опасности можно определить, проведя анализ рисков, составленный на основе

					ПГУ 1.090301.29.001 ПЗ	Лист
						43
Изм.	Лист	№ докум.	Подпись	Дата		

практического опыта. Так, среди главных опасностей для здоровья человека согласно ГОСТу 12.0.003-04 были определены факторы, часть из которых может особенно иметь влияние на исполнителя. Среди них:

- повышенная или пониженная температура/влажность окружающей среды, повышенный уровень шума;
- повышенное напряжение в электросети;
- повышенный уровень электромагнитных излучений - электромагнитное и электростатическое поля, акустический шум, изменение ионного состава воздуха и параметров микроклимата в помещении также могут влиять на процесс работы;
- недостаточная освещенность рабочего места и зрительное напряжение;
- монотонность трудового процесса и нервно-эмоциональные перегрузки - нервно-эмоциональное напряжение при работе за персональным компьютером возникает вследствие дефицита времени, большого объема и плотности информации;
- длительное нахождение в одном положении, зачастую неправильном - повышенные статические и динамические нагрузки у пользователей ПК приводят к жалобам на боли в спине, шейном отделе позвоночника и руках. Из всех недомоганий, обусловленных работой на компьютерах, чаще встречаются те, которые связаны с использованием клавиатуры и неправильном положении при сидячей работе.

2.2 Санитарно-гигиеническая характеристика производства

Одними из важнейших требований при организации рабочего места сотрудника, независимо от его должности, являются обеспечение безопасных, подходящих под конкретного человека, условий для работы, а также недопущение возникновения заболеваний и несчастных случаев на рабочем месте. Общие требования к организации рабочих мест

					ПГУ 1.090301.29.001 ПЗ	Лист
						44
Изм.	Лист	№ докум.	Подпись	Дата		

регулируются Трудовым кодексом, санитарно-эпидемиологическими правилами и нормативами (СанПиН), а также другими правовыми документами.

В настоящее время действуют СанПиН 2.2.4.335916 «Санитарно-эпидемиологические требования к физическим факторам на рабочих местах», утвержденные Постановлением Главного государственного санитарного врача РФ от 21.06.2016 № 81(предназначены для граждан, состоящих в трудовых отношениях, индивидуальных предпринимателей и юридических лиц). Этим документом установлены санитарно-эпидемиологические требования к физическим факторам на рабочих местах и источникам этих физических факторов, а также требования к организации контроля, методам измерения физических факторов на рабочих местах и мерам профилактики вредного воздействия физических факторов на здоровье работающих.

В отношении офисных работников действуют также следующие санитарные нормы и правила:

- при работе с компьютером с плоским монитором рабочее место должно иметь площадь не менее 4,5 кв.м;
- при размещении рабочих мест с ПЭВМ расстояние между рабочими столами с видеомониторами должно быть не менее 2м; размещать рабочие столы следует таким образом, чтобы естественный свет падал на дисплей преимущественно слева;
- экран видеомонитора должен находиться на расстоянии 600 — 700мм от глаз пользователя, но не ближе 500мм с учетом размеров алфавитно-цифровых знаков и символов;
- рабочий стул (кресло) должен быть подъемно-поворотным, регулируемым по высоте и углам наклона сиденья и спинки, а также по расстоянию спинки от переднего края сиденья. При этом регулировка каждого параметра должна быть независимой, легко осуществляемой и иметь надежную фиксацию. При выборе типа рабочего стула (кресла) следует

учитывать рост пользователя, характер и продолжительность работы с ПЭВМ. Поверхности сиденья, спинки и других элементов стула (кресла) должны быть полумягкими, с нескользящим, слабо электризующимся и воздухопроницаемым покрытием, обеспечивающим легкую очистку от загрязнений;

– для сидячих рабочих мест оптимальными параметрами микроклимата на рабочем месте являются: температура воздуха в холодный период года — от 22 до 24 °С, в теплый период года — от 23 до 25 °С; относительная влажность воздуха на постоянных рабочих местах — 40 — 60%, скорость движения воздуха — 0,1м/с.

2.3 Электробезопасность

При рассмотрении вопроса обеспечения электробезопасности разработчика необходимо выделить три основных фактора:

- электроустановки рабочего места программиста;
- вспомогательное электрооборудование;
- окружающая среда помещения.

Помещения, оборудованные вычислительной техникой, как правило, относятся к категории помещений без повышенной опасности так как:

- относительная влажность воздуха не превышает 75%;
- нет токопроводящей пыли и токопроводящих полов;
- температура не превышает длительное время 30 °С;
- отсутствует возможность одновременного прикосновения человека с имеющими соединение с землей металлическими конструкциями.

Таким образом, для предотвращения электротравматизма пользователя, необходимо соблюдать требования безопасности, как при работе с обычной бытовой техникой.

2.4 Пожарная безопасность

Меры пожарной безопасности к помещениям, которые по своему назначению можно отнести к офисам или кабинетам, определяют следующие нормы, правила, изложенные как в законодательной, так и в нормативно-технической базе, например, в Федеральном законе номер 69 «О пожарной безопасности», а также различных ГОСТах, СНиПах, НПБ и СП.

Помещение, в котором ведется разработка программного обеспечения, классифицируется как офисное. К причинам возникновения пожара электрического характера относятся короткие замыкания, перегрузки, большие переходные сопротивления, искрение и электрические дуги, статическое электричество. Основным фактором, от которого зависит пожарная безопасность помещения, является автоматическое средство обнаружения пожаров. В помещении офиса должны быть также установлены: пожарные извещатели автоматической сигнализации, а также иметься воздушно-пенные, воздушно-эмульсионные, порошковые огнетушители.

2.5 Расчетно-аналитическая часть

Расчет освещенности рабочего места сводится к выбору системы освещения, определению необходимого числа светильников, их типа и размещения. Процесс работы программиста происходит в таких условиях, когда естественное освещение недостаточно или отсутствует. Исходя из этого, рассчитаем параметры искусственного освещения.

Расчет освещения производится для комнаты площадью 42 м²: длина - 7 м, ширина - 6 м, высота - 3 м. Равномерное освещение горизонтальной рабочей поверхности достигается при определенных отношениях расстояния

					ПГУ 1.090301.29.001 ПЗ	Лист
						47
Изм.	Лист	№ докум.	Подпись	Дата		

между центрами светильников L , м ($L=1,75*H$) к высоте их подвеса над рабочей поверхностью H , м.

$$L = 1,75 * H = 1,75 * 3 = 5,25 \text{ м.}$$

Число светильников N с люминесцентными лампами (ЛЛ):

$$N = \frac{S}{L * M},$$

где S - площадь помещения, м;

M - расстояние между параллельными рядами, м.

$$N = \frac{6 * 7}{5,25 * 2} = 4.$$

Для достижения равномерной горизонтальной освещенности светильники с ЛЛ рекомендуется располагать сплошными рядами, параллельно стенам с окнами или длинным сторонам помещения.

Для расчета общего равномерного освещения горизонтальной рабочей поверхности используют метод светового потока, учитывающий световой поток, отраженный от потолка и стен.

Для определения количества светильников определим световой поток, падающий на поверхность по формуле:

$$F_p = \frac{E_n * K * S * Z}{N * \mu},$$

где F_p - рассчитываемый световой поток, Лм;

E_n - нормированная минимальная освещенность, Лк (определяется по таблице).

Работу программиста, в соответствии с этой таблицей, можно отнести к разряду точных работ, следовательно, минимальная освещенность будет $E = 300$ Лк при газоразрядных лампах; S - площадь освещаемого помещения (в нашем случае $S = 42 \text{ м}^2$); Z - отношение средней освещенности к минимальной (обычно принимается равным 1.1-1.2 , пусть $Z = 1.1$); K - коэффициент запаса, учитывающий уменьшение светового потока лампы в результате загрязнения светильников в процессе эксплуатации (его значение определяется по таблице коэффициентов запаса для различных помещений и

в нашем случае $K = 1.5$); μ - коэффициент использования, (выражается отношением светового потока, падающего на расчетную поверхность, к суммарному потоку всех ламп и исчисляется в долях единицы; зависит от характеристик светильника, размеров помещения, окраски стен и потолка, характеризуемых коэффициентами отражения от стен (P_c) и потолка (P_n)), значение коэффициентов P_c и P_n определим по таблице зависимостей коэффициентов отражения от характера поверхности: $P_c=30\%$, $P_n=50\%$. Значение μ определим по таблице коэффициентов использования различных светильников. Для этого вычислим индекс помещения по формуле:

$$I = \frac{S}{H(A+B)},$$

где S - площадь помещения, $S = 42 \text{ м}^2$;

h - расчетная высота подвеса, $h = 3 \text{ м}$;

A - ширина помещения, $A = 6 \text{ м}$;

B - длина помещения, $B = 7 \text{ м}$.

Подставив значения получим:

$$I = \frac{42}{3 \cdot (6+7)} = 1,08.$$

Зная индекс помещения I , P_c и P_n , по таблице находим $\mu = 0.3$

Подставим все значения в формулу для определения светового потока F :

$$F = \frac{75 \cdot 1,5 \cdot 42 \cdot 1,1}{4 \cdot 0,3} = 4331,25.$$

Учитывая, что в светильнике по 4 лампы, световой поток одной лампы будет равен. Для освещения выбираем люминесцентные лампы типа ЛБ20, световой поток которых $F = 1180 \text{ Лм}$.

Рассчитаем действительное освещение. В рабочем помещении имеется 4 светильника, в каждом из которых используется по 4 лампы ЛДЦ20. Рассчитаем световой поток, затем рассчитаем значение нормированной освещенности:

$$\Phi = 820 \cdot 4 \cdot 4 = 3280 \text{ Лк};$$

$$E = (\Phi \cdot N \cdot Z) / (S \cdot Z \cdot K) = 227 \text{ Лк}.$$

Полученное значение нормированной освещенности оказалось меньше нормированной минимальной освещенности. Рекомендуется заменить используемые лампы, на лампы с большим световым потоком.

Заключение

В ходе выполнения выпускной квалификационной работы была разработана программная реализация сети пиринговых узлов для выполнения распределенных приложений на языке C++. Сетевое взаимодействие и механизмы сериализации были реализованы с помощью библиотеки Boost.

Разработанное приложение позволяет реализовывать взаимодействие удаленных узлов в вычислительной сети и осуществлять решение определенной задачи на конкретном узле.

Приложение также не привязано к определенной схеме взаимодействия узлов и позволяет проводить вычисления по любому шаблону соединения компьютеров в сети согласно концептуальному графу распределенного приложения при использовании всех дейкстровских конструкций структурного программирования, таких как, последовательность, разветвление, цикл и дополнительных конструкций *fork-join* и *switch-for*.

В дальнейшем можно усовершенствовать разработанное приложение и добавить возможность добавления и удаления узлов на схеме во время выполнения вычислений. Добавить механизмы отказоустойчивости, позволяющие продолжать проводить вычисления при отключении одного из узлов.

Список использованных источников

1. parallel.ru: [Электронный ресурс]. URL: <https://parallel.ru/computers/reviews/meta-computing.html> (дата обращения: 26.04.2022)
2. Олифер В., Олифер Н. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 5-е изд. — СПб.: Питер, 2016. — 992 с.
3. Г.И. Радченко Распределенные вычислительные системы. Учебное пособие – Челябинск: Фотохудожник, 2012. – 184 с.
4. intuit.ru: [Электронный ресурс]. URL: <https://intuit.ru/studies/courses/4115/1230/lecture/24081> (14.05.2022)
5. Б. Амиршахи. Оптимизация Межресурсного Обмена в Распределенных GRID-Вычислениях. – М.: LAP Lambert Academic Publishing, 2012. – 120 с
6. boost.org: [Электронный ресурс]. URL: <https://www.boost.org/> (дата обращения: 27.04.2022)
7. В.П. Галас. Вычислительные системы, сети и телекоммуникации. Учебник в двух частях. Часть 2. Сети и телекоммуникации – ВлГУ, 2017 – 284 с.
8. К.В. Попов, С.А. Зинкин, Н.С. Карамышева. Методы и технологии разработки сетевых приложений для распределенных вычислительных систем: учеб.-метод. пособие – Пенза : ПГУ, 2021. – 226 с.

Приложение А

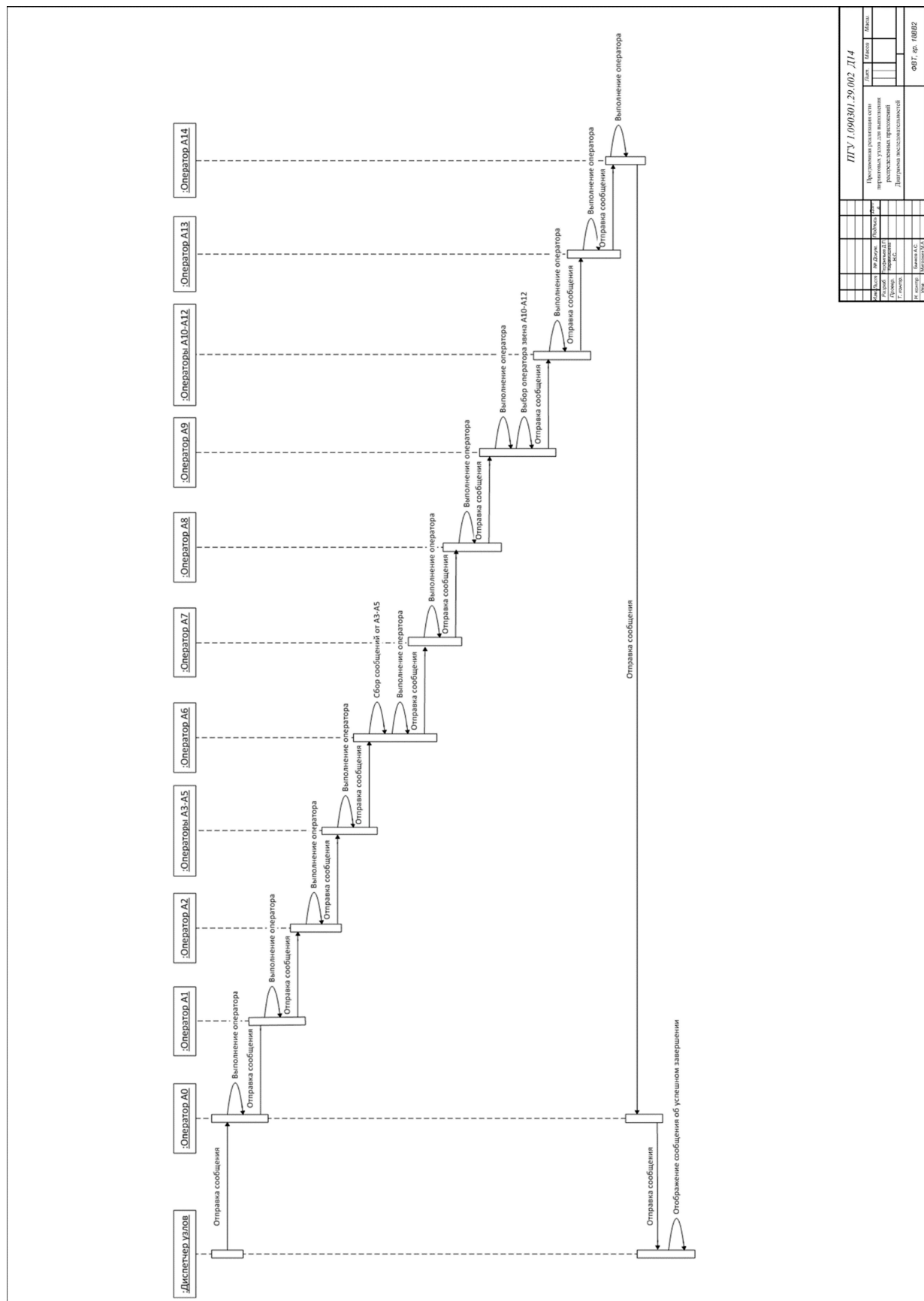
(Обязательное)

Программная реализация сети пиринговых узлов для выполнения
распределенных приложений

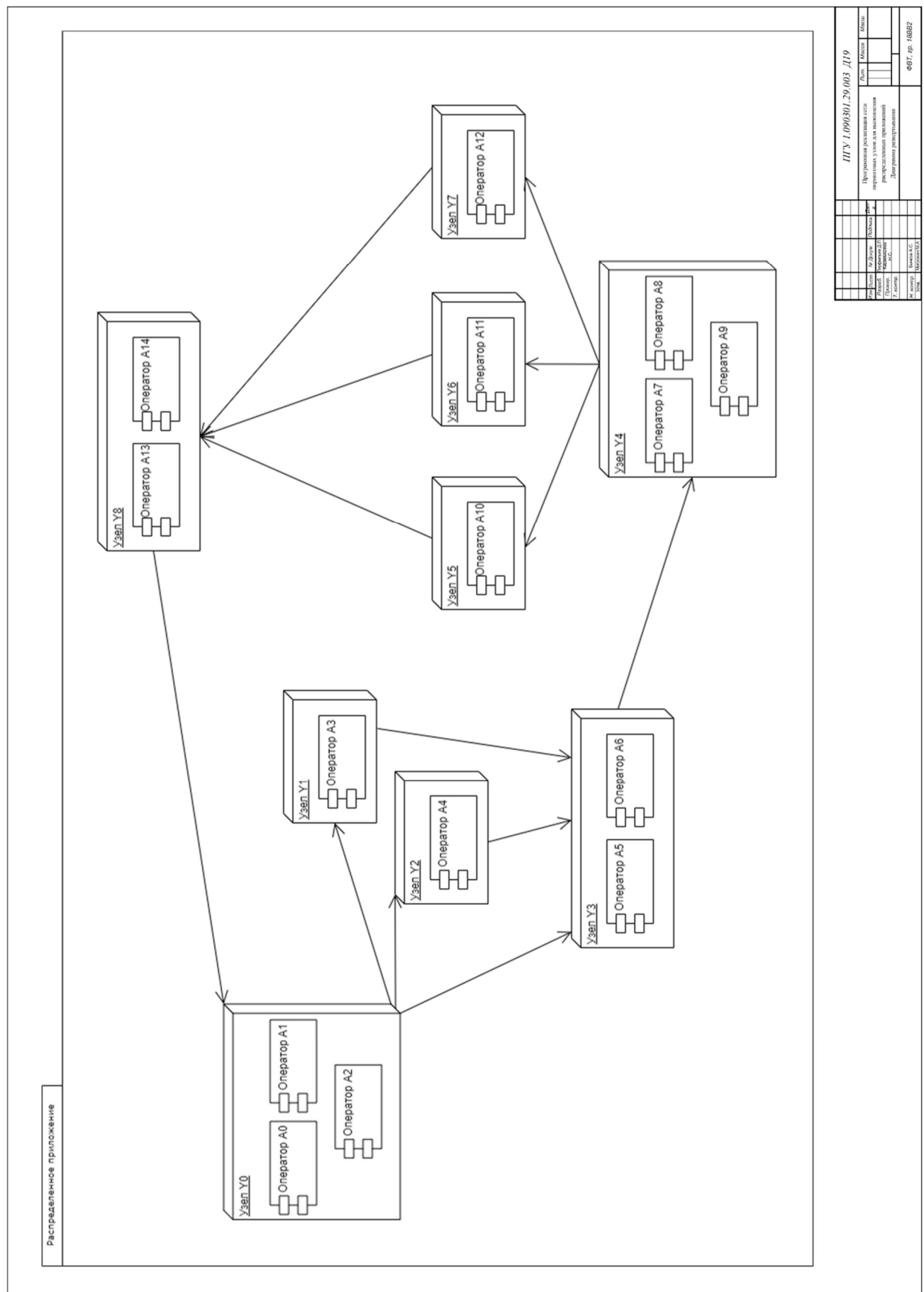
UML – диаграммы

					ПГУ 1.090301.29.001 ПЗ	Лист
						53
Изм.	Лист	№ докум.	Подпись	Дата		

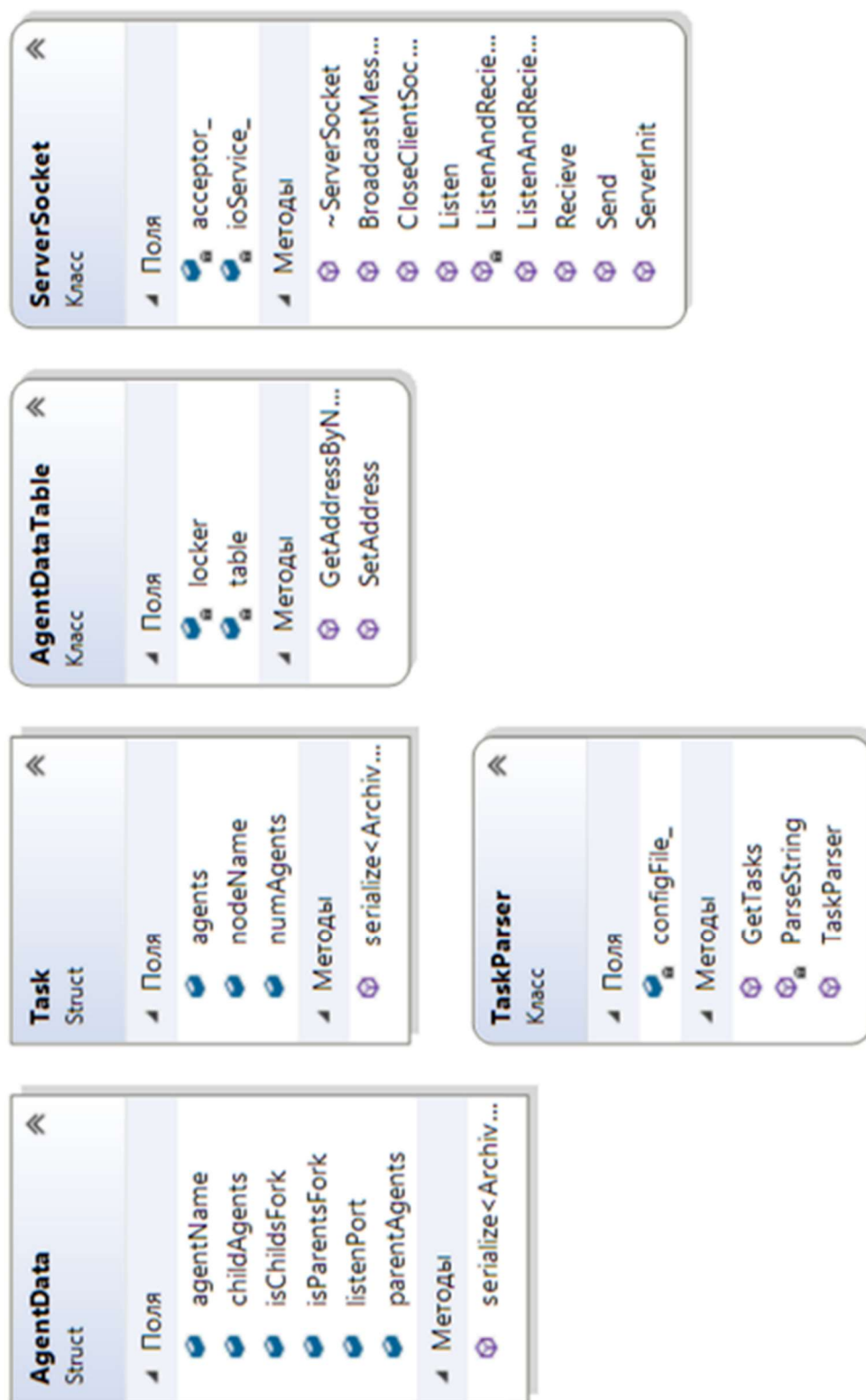
Приложение А.1 – UML – диаграмма последовательностей



Приложение А.2 – UML – диаграмма развертывания



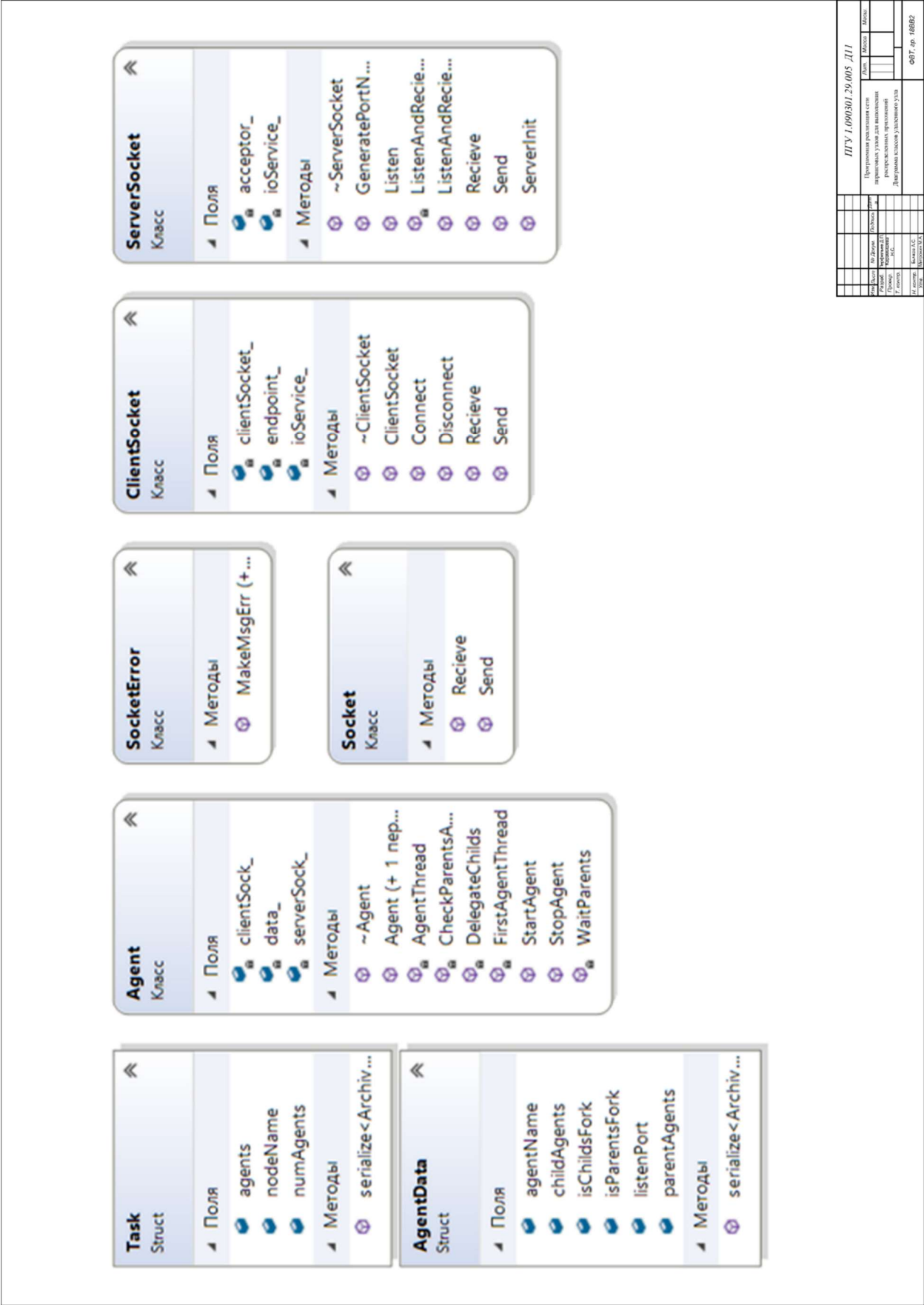
Приложение А.3 – UML – диаграмма классов диспетчера узлов



ПГУ 1.090301.29.004 Д11									
Имя	Фамилия	Имя	Фамилия	Имя	Фамилия	Имя	Фамилия	Имя	Фамилия
Иванов	Иван	Петров	Петр	Сидоров	Сидор	Трофимов	Трофим	Харченко	Харченко
С. Иванов	И. Петров	П. Сидоров	С. Трофимов	Т. Харченко	Х. Харченко	И. Харченко	Х. Харченко	С. Харченко	Х. Харченко
Содержание: Программа разработки сети периферийных устройств для вычисления распределенных приложений. Диаграмма классов диспетчера узлов.									
ФОРМ. № 10002									

Изм.	Лист	№ докум.	Подпись	Дата

Приложение А.4 – UML – диаграмма классов удаленного узла



ПГУ 1.090301.29.005 Д11				
Изм.	Лист	№ докум.	Подпись	Дата
Приложение удаленного узла принадлежит узлу для выполнения распределенных задач (приложение) Диаграмма классов удаленного узла				
ФОРМ. № 10002				

Приложение Б

(Обязательное)

Программная реализация сети пиринговых узлов для выполнения
распределенных приложений

Листинг программы

					ПГУ 1.090301.29.001 ПЗ	Лист
						58
Изм.	Лист	№ докум.	Подпись	Дата		

Файл «AgentDataTable.cpp»

```
#include "AgentDataTable.h"

std::string AgentDataTable::GetAddressByName( const std::string&
name )
{
    locker.lock();

    std::string addr;
    for( auto& agent : table )
    {
        if( std::get<0>( agent ) == name )
        {
            addr = std::get<1>( agent ) + ":" +
std::to_string( std::get<2>( agent ) );
            break;
        }
    }

    locker.unlock();
    return addr;
}

void AgentDataTable::SetAddress( std::tuple<std::string,
std::string, int> agentAddress )
{
    locker.lock();

    table.push_back( agentAddress );

    locker.unlock();
}

Файл «Server.cpp»

#include "TaskParser.h"
```

```

#include "ServerSocket.h"
#include "AgentDataTable.h"
#include <iostream>
#include <future>
#include <sstream>
#include <mutex>

#define CLIENT_PROCESS_PATH "start \"\"
\"C:\\Users\\Иван\\Desktop\\УЧЕБА\\4 курс\\Диплом\\Версия со
временем для
тестов\\Диплом\\Client\\Client\\Debug\\Client.exe\""
#define SERVER_PORT 5555
#define START_AGENT_NAME "A0"

AgentDataTable agentAddressTable;
std::mutex output_locker;
bool isFinish = false;
double time_beg = 0.0;
double time_end = 0.0;

void SaveTime( double totalTime )
{
    std::ofstream fout( "temp.txt" );
    if( fout.is_open() )
    {
        fout << std::to_string( totalTime );
        fout.close();
    }
}

void StartProcesses( int count )
{
    for( int i = 0; i < count; i++ )
    {

```

```

        Sleep( 150 );
        std::system( CLIENT_PROCESS_PATH );
    }
}

void WaitThreads( std::vector<std::future<void>>& results )
{
    try
    {
        int size = results.size();
        int count = 0;

        do
        {
            count = 0;
            for( auto& res : results )
            {
                if( res.valid() )
                {
                    if( res.wait_for(
std::chrono::milliseconds( 200 ) ) == std::future_status::ready
)
                    {
                        res.get();
                        count++;
                    }
                }
            }

            if( isFinish )
                return;

        } while( count != size );
    }
}

```

```

    }
    catch( std::exception ex )
    {
        throw std::exception( ex.what() );
    }
}

std::tuple<std::string, std::string, int> ParseAddress( const
std::string& addr )
{
    std::tuple<std::string, std::string, int> res( "", "", 0 );
    size_t spacePos = addr.find( ' ' );
    size_t pos = addr.find( ':' );

    if( pos != std::string::npos && spacePos !=
std::string::npos )
    {
        std::get<0>( res ) = addr.substr( 0, spacePos );
        std::get<1>( res ) = addr.substr( spacePos + 1, pos -
spacePos - 1 );
        std::get<2>( res ) = atoi( addr.substr( pos + 1
).c_str() );
    }

    return res;
}

void StopClients( std::vector<tcp::socket>& socks, const
std::string& flag = "" )
{
    try
    {
        if(flag != "" )
            ServerSocket::BroadcastMessage( flag, socks );
    }
}

```

```

        ServerSocket::CloseClientSockets( socks );
    }
    catch( std::exception& ex )
    {
        std::cout << "Не удалось разослать клиентам сигнал
остановки - " << flag << std::endl;
    }
}

void InitNode( tcp::socket* socket, Task task )
{
    char* bytes = nullptr;

    try
    {
        std::stringstream ss = Serialization( task );

        ss.seekg( 0, std::ios::end );
        int size = ss.tellg();
        ss.seekg( 0, std::ios::beg );

        bytes = new char[ size ];
        memcpy( bytes, ss.str().c_str(), size );

        ServerSocket::Send( bytes, size, *socket );
        output_locker.lock();
        std::cout << "Задача успешно выслана узлу: " <<
OutputSocketAddress( *socket ) << std::endl;
        output_locker.unlock();

        std::string addr = socket->remote_endpoint().address().to_string();

```

```

        ServerSocket::Send( addr.c_str(), addr.size() + 1,
*socket );

        // ждем от клиента адреса сокетов агентов
        for( int i = 0; i < task.numAgents; i++ )
        {
            std::string address = ServerSocket::Recieve(
*socket );

            std::tuple<std::string, std::string, int> addr =
ParseAddress( address );
            agentAddressTable.SetAddress( addr );
        }

        delete[] bytes;
    }
    catch( std::exception& ex )
    {
        std::string error = "Ошибка инициализации узла: " +
task.nodeName + "\n" + std::string( ex.what() );

        delete[] bytes;
        throw std::exception( error.c_str() );
    }
}

void AgentHandler( tcp::socket* agentSock, bool isStartThread )
{
    try
    {
        if( isStartThread )
        {

            //std::cout << "Нажмите любую клавишу для запуска
процесса вычислений..." << std::endl;

```

```

        //std::getchar();
        ServerSocket::Send( "start", 6, *agentSock );
        time_beg = clock();
    }

while( true )
{
    std::string flag;
    std::string op;
    size_t pos;

    std::string request = ServerSocket::Recieve(
*agentSock );

    if( request == "finish" )
    {
        isFinish = true;
        continue;
    }

    if( ( pos = request.find( ';' ) ) !=
std::string::npos )
    {
        flag = request.substr( 0, pos );
        op = request.substr( pos + 1 );
    }

    if( flag == "error" )
    {
        std::string error = "Агент " + op + "
завершился с ошибкой\n";
        throw std::exception( error.c_str() );
    }
    else if( flag == "success" )
    {

```

```

        output_locker.lock();
        std::cout << "Агент " << op << " завершился
успешно\n";

        output_locker.unlock();
        break;
    }
    else
    {
        std::string addr =
agentAddressTable.GetAddressByName( request );
        ServerSocket::Send( addr.c_str(), addr.size()
+ 1, *agentSock );
    }
}

catch( std::exception& ex )
{
    throw std::exception( ex.what() );
}
}

int main()
{
    SetConsoleOutputCP( 1251 );

    std::string exitCode = "success";
    ServerSocket server;
    std::vector<Task> tasks;
    std::vector<tcp::socket> nodeSocks;
    std::vector<tcp::socket> agentSocks;
    std::vector<std::future<void>> results;

    try
    {

```

```

std::cout << "Сервер запущен" << std::endl;

TaskParser parser( "config.txt" );
tasks = parser.GetTasks();
std::cout << "Конфигурационный файл успешно разобран"
<< std::endl;

// инициализация узлов
std::cout << "Прослушивание подключений..." <<
std::endl;
server.ServerInit( SERVER_PORT );
//std::async( std::launch::async, StartProcesses,
tasks.size() );

for( int i = 0; i < tasks.size(); i++ )
{
    nodeSocks.push_back( server.Listen() );
    std::cout << "Новый узел подключен - " <<
OutputSocketAddress( nodeSocks[ i ] ) << " - " <<
tasks[i].nodeName << std::endl;
}
std::cout << "Требуемое число клиентов подключено\n" <<
std::endl;

std::cout << "Рассылка задач узлам..." << std::endl;
for( int i = 0; i < nodeSocks.size(); i++ )
    results.push_back( std::async( std::launch::async,
InitNode, &nodeSocks[ i ], tasks[ i ] ) );

WaitThreads( results );

// конец инициализации узлов. если get не бросил
исключений значит таблица адресов заполнена

```

```

std::cout << "Все задачи успешно разосланы узлам\n" <<
std::endl;

// начало работы с агентами
results.clear();
std::cout << "Работа агентов..." << std::endl;
int count = 0;
for( auto& task : tasks )
{
    for( int i = 0; i < task.numAgents; i++ )
    {
        agentSocks.push_back( server.Listen() );
        std::cout << "Новый агент подключен - " <<
OutputSocketAddress( agentSocks[ count ] ) << " - " <<
task.agents[ i ].agentName << std::endl;
        count++;
    }
}

//Sleep( 200 );
count = 0;
for( auto& task : tasks )
{
    for( int i = 0; i < task.numAgents; i++ )
    {
        bool isStartThread = false;
        if( task.agents[ i ].agentName ==
START_AGENT_NAME )
        {
            isStartThread = true;
        }
    }
}

```

```

        results.push_back( std::async(
std::launch::async, AgentHandler, &agentSocks[ count ],
isStartThread ) );

        count++;

    }

}

WaitThreads( results );
time_end = clock();

}

catch( std::exception& ex )
{

    std::cout << ex.what() << std::endl;
    exitCode = "error";

}

if( isFinish )
{

    double res = ( time_end - time_beg ) / CLOCKS_PER_SEC;
    SaveTime( res );

}

if( exitCode == "error" )
    std::cout << "Работа агентов завершена с ошибкой" <<
std::endl;
else
    std::cout << "Работа агентов успешно завершена" <<
std::endl;

//std::cout << "Нажмите любую клавишу для завершения работы
сервера...";

//std::getchar();

StopClients( nodeSocks, exitCode );

```

```

        StopClients( agentSocks );
    }

```

Файл «ServerSocket.cpp»

```

#include "ServerSocket.h"
#include <future>
#include <string>

// прослушку прекратить через socket.close

void ServerSocket::ServerInit( int port )
{
    acceptor_ = new tcp::acceptor( ioService_, tcp::endpoint(
tcp::v4(), port ) );
}

ServerSocket::~ServerSocket()
{
    if( acceptor_->is_open() )
        acceptor_->close();

    delete acceptor_;
    acceptor_ = 0;
}

std::string ServerSocket::ListenAndRecieve()
{
    tcp::socket client = Listen();
    return Recieve( client );
}

std::future<std::string> ServerSocket::ListenAndRecieveAsync()
{

```

```

        return std::async( std::launch::async,
&ServerSocket::ListenAndRecieve, this );
    }

tcp::socket ServerSocket::Listen()
{
    try
    {
        tcp::socket clientSocket( ioService_ );
        acceptor_>accept( clientSocket );
        return std::move( clientSocket );
    }
    catch( std::exception& ex )
    {
        std::string error = "Ошибка прослушивания новых
подключений" + std::string( ex.what() );
        throw std::exception( error.c_str() );
    }
}

int ServerSocket::Send( const char* buf, int size, tcp::socket&
socket )
{
    int part, len = 0;

    if( !socket.is_open() )
        return -1;

    try
    {
        socket.send( asio::buffer( &size, sizeof( int ) ) );
        const char* last = buf + size;

        while( buf < last )
        {

```

```

        part = socket.send( asio::buffer( buf, size ) );
        buf += part;
        len += part;
    }
}

catch( std::exception& ex )
{
    std::string error = "Ошибка передачи данных: "
        + socket.remote_endpoint().address().to_string()
        + ":"
        + std::to_string( socket.remote_endpoint().port()
)
        + "\n"
        + std::string( ex.what() );

    throw std::exception( error.c_str() );
}

return len;
}

std::string ServerSocket::Recieve( tcp::socket& socket )
{
    int size, part;
    char* inputBuf;
    std::string buf;

    if( !socket.is_open() )
        return std::string();

    try
    {
        socket.receive( asio::buffer( &size, sizeof( int ) ) );
        inputBuf = new char[ size ];
    }
}

```

```

char* tempPointer = inputBuf;
char* last = inputBuf + size;

while( tempPointer < last )
{
    part = socket.receive( asio::buffer( tempPointer,
size ) );
    tempPointer += part;
}

buf = inputBuf;
}
catch( std::exception& ex )
{
    std::string error = "Ошибка приема данных от: "
        + socket.remote_endpoint().address().to_string()
        + ":"
        + std::to_string( socket.remote_endpoint().port()
)
        + "\n"
        + std::string( ex.what() );

    throw std::exception( error.c_str() );
}

delete[] inputBuf;
return buf;
}

void ServerSocket::BroadcastMessage( const std::string& msg,
std::vector<tcp::socket>& socks )
{
    int count = 0;
    try

```

```

    {
        for( auto& socket : socks )
        {
            if( socket.is_open() )
            {
                Send( msg.c_str(), msg.size() + 1, socket );
                count++;
            }
        }
    }
    catch( std::exception& ex )
    {
        std::string error = "Рассылка данных была прервана.
Разослано сообщений: "
            + std::to_string( count ) + " из " +
std::to_string( socks.size() )
            + "\n" + ex.what();

        throw std::exception( error.c_str() );
    }
}

void ServerSocket::CloseClientSockets( std::vector<tcp::socket>&
socks )
{
    for( auto& socket : socks )
    {
        socket.close();
    }

    socks.clear();
}

```

Файл «TaskParser.cpp»

```

#include "TaskParser.h"
#include <sstream>
#define COUNT_SEPARATORS_CONFIG 4

TaskParser::TaskParser( const std::string& configFilepath )
{
    configFile_.open( configFilepath );
    if( !configFile_.is_open() )
    {
        throw std::exception( "Не удалось открыть
конфигурационный файл" );
    }
}

std::vector<Task> TaskParser::GetTasks()
{
    std::vector<Task> tasks;
    std::string stringTask;

    while( std::getline( configFile_, stringTask ) )
    {
        AgentData agent;
        std::string nodeName;
        bool createNew = true;

        if( !ParseString( stringTask, agent, nodeName ) )
        {
            throw std::exception( "Не удалось разобрать
конфигурационный файл" );
        }

        for( auto& task : tasks )
        {
            if( task.nodeName == nodeName )

```

```

        {
            task.numAgents++;
            task.agents.push_back( agent );
            createNew = false;
            break;
        }
    }

    if( createNew )
    {
        Task task;
        task.agents.push_back( agent );
        task.nodeName = nodeName;
        task.numAgents = 1;

        tasks.push_back( task );
    }
}

return tasks;
}

bool TaskParser::ParseString( const std::string& strTask,
AgentData& agent, std::string& nodeName )
{
    auto ParseComplexElem = []( const std::string& complexElem,
std::vector<std::string>& ops, bool& isFork )
    {
        std::istringstream iss( complexElem );
        std::string tempStr;

        if( complexElem.find( '/' ) != std::string::npos )
        {
            isFork = true;

```

```

        while( std::getline( iss, tempStr, '/' ) )
        {
            ops.push_back( tempStr );
        }
    }
    else if( complexElem.find( ',' ) != std::string::npos )
    {
        isFork = false;
        while( std::getline( iss, tempStr, ',' ) )
        {
            ops.push_back( tempStr );
        }
    }
    else
    {
        ops.push_back( complexElem );
        isFork = false;
    }
};

```

```

std::istringstream iss( strTask );
std::string tmpStr;
std::vector<std::string> elems;

```

```

while( std::getline( iss, tmpStr, ';' ) )
{
    elems.push_back( tmpStr );
}

```

```

if( elems.size() != COUNT_SEPARATORS_CONFIG )
    return false;

```

```

for( auto& elem : elems )
{

```

```

        if( elem.empty() )
            return false;
    }

    agent.agentName = elems[ 0 ];
    nodeName = elems[ 1 ];

    bool isFork;
    std::vector<std::string> ops;

    ParseComplexElem( elems[ 2 ], ops, isFork );
    agent.isParentsFork = isFork;
    agent.parentAgents = ops;
    ops.clear();

    ParseComplexElem( elems[ 3 ], ops, isFork );
    agent.isChildsFork = isFork;
    agent.childAgents = ops;

    return true;
}

```

Файл «Agent.cpp»

```

#include "Agent.h"
#include "ClientSocket.h"
#include "ServerSocket.h"
#include <iostream>
#include <thread>
#include <exception>
#include <string>
#include <future>
#include <mutex>

namespace agent

```

```

{

std::mutex locker;

static void SyncOut( const std::string& output )
{
    locker.lock();
    std::cout << output;
    locker.unlock();
}

static void SyncOutData( const std::string& msg,
std::vector<std::string>& data )
{
    locker.lock();
    std::cout << msg;
    for( auto& one : data )
    {
        std::cout << one << " ";
    }
    std::cout << "\n";
    locker.unlock();
}

void Agent::DelegateChilds( ClientSocket& socket,
std::vector<std::pair<std::string, int>>& childs, const
std::string& data )
{
    auto AsyncDelegate = [ &socket ]( const std::string& ip, int
port, const std::string& data )
    {
        socket.Disconnect();
        socket.Connect( ip, port );
        socket.Send( data.c_str(), data.length() + 1 );
    }
}

```

```

};

try
{
    if( data_.isChildsFork )
    {
        for( auto& child : childs )
        {
            AsyncDelegate( child.first, child.second,
data );
        }
    }
    else
    {
        int random = 0 + rand() %
data_.childAgents.size();
        AsyncDelegate( childs[ random ].first, childs[
random ].second, data );
    }
}
catch( std::exception& e )
{
    std::string error = "Ошибка отправки сигнала дочерним
агентам\n" + std::string( e.what() );
    throw std::exception( error.c_str() );
}

std::vector<std::string> Agent::WaitParents( ServerSocket&
socket, int port, int parentsCount )
{
    try
    {
        socket.ServerInit( port );

```

```

std::vector<std::future<std::string>> returnValues;
std::vector<std::string> parentsDatas;

if( data_.isParentsFork )
{
    for( int i = 0; i < parentsCount; i++ )
    {
        returnValues.push_back(
socket.ListenAndRecieveAsync() );
    }
}
else
{
    returnValues.push_back(
socket.ListenAndRecieveAsync() );
}

for( auto& value : returnValues )
{
    parentsDatas.push_back( value.get() );
}

return parentsDatas;
}
catch( std::exception& e )
{
    std::string error = "Ошибка получения сигнала от
родительского агента\n" + std::string( e.what() );
    throw std::exception( error.c_str() );
}
}

void Agent::CheckParentsAccuracy( std::vector<std::string>&
recievedParents )

```

```

{
    int count = 0;
    for( auto& recievedParent : recievedParents )
    {
        for( auto& expectedParent : data_.parentAgents )
        {
            if( recievedParent == expectedParent )
            {
                count++;
                break;
            }
        }
    }

    // если родительских агентов не несколько одновременно, то
    хватит и одного совпадения
    if( !data_.isParentsFork && count == 1 )
        return;

    if( count != data_.parentAgents.size() )
    {
        throw std::exception( "Ошибка получения данных от
родительских узлов\n" );
    }
}

void Agent::StopAgent()
{
    delete serverSock_;
    delete clientSock_;

    serverSock_ = 0;
    clientSock_ = 0;
}

```

```

void Agent::StartAgent( std::function<void( const std::string&
)> task, bool isFirstAgent )
{
    if( isFirstAgent )
        std::thread( &Agent::FirstAgentThread, this, task
).detach();
    else
        std::thread( &Agent::AgentThread, this, task
).detach();
}

void RequestAgentsAddress( ClientSocket* socket, const
std::vector<std::string>& agents,
std::vector<std::pair<std::string, int>>& addrs )
{
    for( int i = 0; i < agents.size(); i++ )
    {
        std::string agentName = agents[ i ];
        socket->Send( agentName.c_str(), agentName.size() + 1
);

        std::string response = socket->Recieve();

        std::pair<std::string, int> agentAddr;
        agentAddr.first = response.substr( 0, response.find(
':' ) );
        agentAddr.second = atoi( response.substr(
response.find( ':' ) + 1 ).c_str() );

        addrs.push_back( agentAddr );
    }
}

```

```

void Agent::FirstAgentThread( std::function<void( const
std::string& )> task )
{
    std::string msg = "success;" + data_.agentName;
    ClientSocket agent;

    try
    {
        agent.Connect( SERVER_IP, SERVER_PORT );

        SyncOut( "[ " + data_.agentName + " ] Ожидание сигнала
о начале вычислений от диспетчер узлов\n" );
        agent.Reclieve();

        SyncOut( "[ " + data_.agentName + " ] Сигнал старта
получен\n" );

        task( data_.agentName );

        std::vector<std::pair<std::string, int>> nxtAgents;
        RequestAgentsAddress( &agent, data_.childAgents,
nxtAgents );

        SyncOut( "[ " + data_.agentName + " ] Отправка данных
дочернем узлам\n" );
        clientSock_ = new ClientSocket();
        DelegateChilds( *clientSock_, nxtAgents,
data_.agentName );

        SyncOutData( "[ " + data_.agentName + " ] Данные
успешно отправлены дочерним узлам: ", data_.childAgents );

        SyncOutData( "[ " + data_.agentName + " ] Ожидание
данных от родительских узлов: ", data_.parentAgents );

```

```

        serverSock_ = new ServerSocket();
        std::vector<std::string> parentsDatas = WaitParents(
*serverSock_, data_.listenPort, data_.parentAgents.size() );
        CheckParentsAccuracy( parentsDatas );

        SyncOutData( "[ " + data_.agentName + " ] Данные
получены: ", parentsDatas );

        agent.Send( "finish", 7 );
    }
    catch( const std::exception& ex )
    {
        std::cout << "Ошибка работы агента " << data_.agentName
<< std::endl << ex.what() << std::endl;
        msg = "error;" + data_.agentName;
    }

    agent.Send( msg.c_str(), msg.size() + 1 );
    agent.Disconnect();
}

void Agent::AgentThread( std::function<void( const std::string&
)> task )
{
    std::string msg = "success;" + data_.agentName;
    ClientSocket agentSocket;

    try
    {
        Sleep( 200 );
        agentSocket.Connect( SERVER_IP, SERVER_PORT );//
        подключаемся к серверу)

```

```

        SyncOutData( "[ " + data_.agentName + " ] Ожидание
данных от родительских узлов: ", data_.parentAgents );

        serverSock_ = new ServerSocket();
        std::vector<std::string> parentsDatas = WaitParents(
*serverSock_, data_.listenPort, data_.parentAgents.size() );
        CheckParentsAccuracy( parentsDatas );

        SyncOutData( "[ " + data_.agentName + " ] Данные
получены: ", parentsDatas );

        task( data_.agentName );

        // получим у сервера адрес след агента
        std::vector<std::pair<std::string, int>> nxtAgents;
        RequestAgentsAddress( &agentSocket, data_.childAgents,
nxtAgents );

        if( data_.agentName == "A2" )
            int ll = 0;

        SyncOut( "[ " + data_.agentName + " ] Отправка данных
дочернем узлам\n" );
        clientSock_ = new ClientSocket();
        DelegateChilds( *clientSock_, nxtAgents,
data_.agentName );

        SyncOutData( "[ " + data_.agentName + " ] Данные
успешно отправлены дочерним узлам: ", data_.childAgents );
    }
    catch( const std::exception& ex )
    {
        std::cout << "Ошибка работы агента " << data_.agentName
<< std::endl << ex.what() << std::endl;

```

```

        msg = "error;" + data_.agentName;
    }

    agentSocket.Send( msg.c_str(), msg.size() + 1 );
    agentSocket.Disconnect();
}

```

Файл «Client.cpp»

```

#include "Agent.h"
#include <iostream>
#include <vector>
#include <sstream>

void AgentTask( const std::string& agentName )
{
    std::cout << "Оператор " << agentName << " успешно выполнен"
    << std::endl;
}

int main()
{
    SetConsoleOutputCP( 1251 );

    using namespace agent;

    int agentPort = 5555;
    std::string signal;

    try
    {
        //запуск узла и коннект с сервером
        ClientSocket nodeSocket;
        nodeSocket.Connect( SERVER_IP, SERVER_PORT );
    }
}

```

```

        std::cout << "Соединение с диспетчером узлов
установлено\n\nОжидание задачи от диспетчера узлов" <<
std::endl;

        std::string strTask = nodeSocket.Recieve();
        std::stringstream ss( strTask );
        Task task = Deserialization<Task>( std::move( ss ) );
        std::cout << "Задача получена\n" << std::endl;

        std::cout << "Узел: " << task.nodeName << std::endl;
        std::cout << "Операторы узла: ";
        for( auto& op : task.agents )
        {
            std::cout << op.agentName << " ";
        }
        std::cout << "\n\n";

        // исключительно для работы на одной машине,
        // для реальной сети условие не требуется
        //-----
        int pos = task.nodeName.find( 'Y' );
        int kf = atoi( task.nodeName.substr( pos + 1 ).c_str()
);

        agentPort += 30 * kf;
        //-----

        // отправим порты агентов на сервер
        std::string ip = nodeSocket.Recieve();
        for( int i = 0; i < task.numAgents; i++ )
        {
            agentPort = ServerSocket::GeneratePortNumber( ip,
agentPort );

            std::string agentAddr = task.agents[ i ].agentName
+ " " + ip + ":" + std::to_string( agentPort );

```

```

        nodeSocket.Send( agentAddr.c_str(),
agentAddr.size() + 1 );
        task.agents[ i ].listenPort = agentPort;
    }

    Sleep( 500 );
    // создание агентов
    std::vector<Agent> agents;
    for(int i = 0; i < task.numAgents; ++i )
    {
        agents.push_back( std::move( Agent( task.agents[ i
] ) ) );
    }

    Sleep( 200 );
    for( int i = 0; i < task.numAgents; i++ )
    {
        bool isStartAgent = false;

        if( task.agents[ i ].agentName == "A0" )
            isStartAgent = true;

        agents[ i ].StartAgent( AgentTask, isStartAgent );
    }

    signal = nodeSocket.Recieve();
}
catch( const std::exception& e )
{
    std::cout << e.what() << std::endl;
    signal = "error";
}

if( signal == "error" )

```

```

        std::cout << "Целостность сети нарушена. Программа
завершилась аварийно." << std::endl;
    else
        std::cout << "Вычисления завершены успешно." <<
std::endl;

    //std::cout << "Нажмите любую клавишу для закрытия
приложения...";
    //std::getchar();
}

```

Файл «ClientSocket.cpp»

```

#include "ClientSocket.h"
#include "Error.h"

void ClientSocket::Connect( const std::string& ip, int port )
{
    try
    {
        if( !clientSocket_>is_open() )
        {
            endpoint_.address(
asio::ip::address_v4::from_string( ip ) );
            endpoint_.port( port );
            //clientSocket_>open( tcp::v4() );
            clientSocket_>connect( endpoint_ );
        }
    }
    catch( std::exception& ex )
    {
        SocketError::MakeMsgErr( "Не удалось подключиться к
конечной точке", ex.what(), endpoint_ );
    }
}

```

```

int ClientSocket::Send( const char* buf, int size )
{
    int bytes = 0;
    try
    {
        if( clientSocket_>is_open() )
        {
            bytes = socket_write_read::Socket::Send( buf,
size, *clientSocket_ );
        }
    }
    catch( std::exception& ex )
    {
        SocketError::MakeMsgErr( "Ошибка передачи данных узлу",
ex.what(), endpoint_ );
    }

    return bytes;
}

std::string ClientSocket::Recieve()
{
    std::string buffer;
    try
    {
        if( clientSocket_>is_open() )
        {
            buffer = socket_write_read::Socket::Recieve(
*clientSocket_ );
        }
    }
    catch( std::exception& ex )
    {

```

```

        SocketError::MakeMsgErr( "Ошибка приема данных от
узла", ex.what(), endpoint_ );
    }

    return buffer;
}

void ClientSocket::Disconnect()
{
    if( clientSocket_>is_open() )
    {
        clientSocket_>close();
    }
}

ClientSocket::~ClientSocket()
{
    Disconnect();
    delete clientSocket_;
    clientSocket_ = 0;
}

```

Файл «ServerSocket.cpp»

```

#include "ServerSocket.h"
#include "Error.h"
#include <future>
#include <string>

int ServerSocket::GeneratePortNumber( const std::string& ip,
unsigned short startNum )
{
    asio::io_service io_service;
    tcp::socket tempSocket( io_service );
    tcp::endpoint ep;

```

```

boost::system::error_code ec;

ep.address( asio::ip::address_v4::from_string( ip ) );
tempSocket.open( tcp::v4() );

if( tempSocket.is_open() )
{
    do
    {
        startNum++;
        ep.port( startNum );
        tempSocket.bind( ep, ec );
    } while( ec.value() != boost::system::errc::success );

    tempSocket.close();
}
else
{
    throw std::exception( "Невозможно занять порт для
сокета агента" );
}

return startNum;
}

void ServerSocket::ServerInit( int port )
{
    try
    {
        acceptor_ = new tcp::acceptor( ioService_,
tcp::endpoint( tcp::v4(), port ) );
    }
    catch( std::exception& ex )
    {

```

```

        SocketError::MakeMsgErr( "Не удалось запустить сервер",
ex.what() );
    }
}

ServerSocket::~ServerSocket()
{
    if( acceptor_>is_open() )
        acceptor_>close();

    delete acceptor_;
    acceptor_ = 0;
}

std::string ServerSocket::ListenAndRecieve()
{
    tcp::socket client = Listen();
    return Recieve( client );
}

std::future<std::string> ServerSocket::ListenAndRecieveAsync()
{
    return std::async( std::launch::async,
&ServerSocket::ListenAndRecieve, this );
}

tcp::socket ServerSocket::Listen()
{
    try
    {
        tcp::socket clientSocket( ioService_ );
        acceptor_>accept( clientSocket );
        return std::move( clientSocket );
    }
}

```

```

        catch( std::exception& ex )
        {
            SocketError::MakeMsgErr( "Ошибка прослушивания новых
подключений", ex.what() );
        }
    }

int ServerSocket::Send( const char* buf, int size, tcp::socket&
client )
{
    int bytes = 0;
    try
    {
        if( client.is_open() )
        {
            bytes = socket_write_read::Socket::Send( buf,
size, client );
        }
    }
    catch( std::exception& ex )
    {
        SocketError::MakeMsgErr( "Ошибка передачи данных узлу",
ex.what(), client.remote_endpoint() );
    }

    return bytes;
}

std::string ServerSocket::Recieve( tcp::socket& client )
{
    std::string buffer;
    try
    {
        if( client.is_open() )

```

```

        {
            buffer = socket_write_read::Socket::Recieve(
client );
        }
    }
    catch( std::exception& ex )
    {
        SocketError::MakeMsgErr( "Ошибка приема данных от
узла", ex.what(), client.remote_endpoint() );
    }

    return buffer;
}

```

Файл «TestServer.cpp»

```

#include <iostream>
#include <boost/asio.hpp>
#include <boost/filesystem.hpp>
#include <windows.h>
#include <vector>
#include <future>

#define PROCESS_NAME "Server.exe"
#define PORT 7777

using namespace boost;
using namespace boost::asio;

void InitTest( int& numNodes, int& numLaunches )
{
    std::cout << "Введите количество узлов в сети: ";
    std::cin >> numNodes;

    std::cout << "Введите желаемое количество запусков: ";

```

```

        std::cin >> numLaunches;
    }

    std::vector<ip::tcp::socket> ListenClients( ip::tcp::acceptor&
    acceptor, const ip::tcp::endpoint& ep, int numNodes )
    {
        std::vector<ip::tcp::socket> clientSocks;

        acceptor.open( ep.protocol() );
        acceptor.set_option( socket_base::reuse_address( true ) );
        acceptor.bind( ep );
        acceptor.listen();

        for( int i = 0; i < numNodes; ++i )
        {
            clientSocks.push_back( std::move( acceptor.accept() ) )
        };

        return std::move( clientSocks );
    }

    std::string GetDispetcherFilePath()
    {
        int size = 1024;
        char path[ 1024 ];

        GetModuleFileName( NULL, path, size );
        std::string strPath( path );

        filesystem::path dirPath( strPath );
        dirPath = dirPath.branch_path();
        dirPath /= PROCESS_NAME;
    }

```

```

        return dirPath.string();
    }

void Broadcast( std::vector<ip::tcp::socket>& socks, const
std::string& msg )
{
    for( auto& sock : socks )
    {
        sock.send( boost::asio::buffer( msg.c_str(), msg.size()
+ 1 ) );
    }
}

int StartNodeDispatcher( std::string path )
{
    DWORD exitCode;
    STARTUPINFO si;
    PROCESS_INFORMATION piCom;

    ZeroMemory( &si, sizeof( STARTUPINFO ) );
    si.cb = sizeof( STARTUPINFO );

    if( CreateProcess( path.c_str(), NULL, NULL, NULL, FALSE,
CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom ) == 0 )
    {
        return -1;
    }

    WaitForSingleObject( piCom.hProcess, INFINITE );
    GetExitCodeProcess( piCom.hProcess, &exitCode );

    return exitCode;
}

```

```

int main()
{
    SetConsoleCP( 1251 );
    SetConsoleOutputCP( 1251 );

    int numNodes, numLaunches;
    double totalTime = 0.0;
    bool success = false;
    std::string startSignal = "begin";
    std::string endSingal = "end";

    io_service ioService;
    std::vector<ip::tcp::socket> clientSocks;
    ip::tcp::endpoint ep( ip::tcp::v4(), PORT );
    ip::tcp::acceptor acceptor( ioService );

    InitTest( numNodes, numLaunches );

    std::cout << "Ожидание подключения тестовых клиентов..." <<
std::endl;

    try
    {
        clientSocks = std::move( ListenClients( acceptor, ep,
numNodes ) );
        std::string filepath = GetDispetcherFilePath();

        std::cout << "Тестовые клиенты подключены" <<
std::endl;

        for( int i = 0; i < numLaunches; ++i )
        {
            std::ofstream fout( "temp.txt",
std::ios_base::trunc | std::ios_base::out );

```

```

        if( fout.is_open() )
            fout.close();
        else
            break;

        std::future<int> exitCodeProcess = std::async(
std::launch::async, StartNodeDispatcher, filepath );

        Broadcast( clientSocks, startSignal );

        if( exitCodeProcess.get() != 0 )
        {
            break;
        }

        std::ifstream fin( "temp.txt" );
        if( fin.is_open() )
        {
            std::string timeStr;
            std::getline( fin, timeStr );

            std::cout << "Время работы сети при " <<
std::to_string( i ) << " запуске - " << timeStr << " секунд" <<
std::endl;

            totalTime += atof( timeStr.c_str() );
        }
        else
        {
            break;
        }
    }

    success = true;

```

```

    }

    catch( std::exception ex )
    {
        std::cout << "Некорректное завершение тестов" <<
std::endl;

        exit( -1 );
    }

    if( success )
    {
        double res = totalTime / numLaunches;

        std::cout << "Среднее время работы сети при " <<
std::to_string( numLaunches ) << " запусках - " <<
std::to_string( res ) << " секунд" << std::endl;
    }

    Broadcast( clientSocks, endSingal );

    for( auto& sock : clientSocks )
    {
        if( sock.is_open() )
            sock.close();
    }

    if( acceptor.is_open() )
        acceptor.close();

    std::cout << "Корректное завершение тестов" << std::endl;

    std::getchar();
}

```

Приложение В

(Обязательное)

Программная реализация сети пиринговых узлов для выполнения
распределенных приложений

Презентация

					ПГУ 1.090301.29.001 ПЗ	Лист
						102
Изм.	Лист	№ докум.	Подпись	Дата		

Программная реализация сети пиринговых узлов для выполнения распределенных приложений

Выполнил:

студент группы 18BB2

Перфильев Д.П.

Руководитель:

к.т.н., доцент каф. ВТ ПГУ Карамышева Н.С.

Цели и задачи

Данная исследовательская работа преследует следующие цели:

- ▶ разработка шаблона для лабораторных заданий по дисциплинам, связанным с созданием распределенных сетевых приложений;
- ▶ разработка технологии создания распределенных приложений на основании представления работы узла сети, как узла пиринговой системы;
- ▶ получение квалифицированных навыков создания сложных сетевых приложений, реализующих произвольный распределенный алгоритм.

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090301.29.001 ПЗ

Лист

103

Задачи, которые должно решать разрабатываемое приложение:

- ▶ создание удобного интерфейса пользователя, представленного консольным терминалом;
- ▶ реализация алгоритмов передачи данных между узлами;
- ▶ разработка алгоритмов, обеспечивающих возможность проведения вычислений на узлах вычислительной сети параллельно.

Модель распределенной обработки данных

Пиринговая (*P2P* - peer-to-peer networking) технология - это технология организации одноранговых сетей.
Пиринговая организация работы группы пользователей основана на понятии «равноправного участника».



P2P-network

Изм.	Лист	№ докум.	Подпись	Дата

Обзор аналогов

JADE - это программное обеспечение промежуточного слоя (см. рис.1), разработанное компанией TILAB, предназначенное для создания распределенных мультиагентных приложений на основе транспортной архитектуры «точка-точка».



- ▶ *BOINC* — программный комплекс для быстрой организации распределённых вычислений. Состоит из серверной и клиентской частей.
- ▶ *LHC Computing Grid* — гид, спроектированный в *CERN* и предназначенный для обработки больших объёмов данных, поступающих с большого адронного коллайдера.

Используемые программные средства

- ▶ Язык программирование - C++
- ▶ Собрание библиотек классов для C++ - Boost
- ▶ Протокол передачи данных TCP/IP

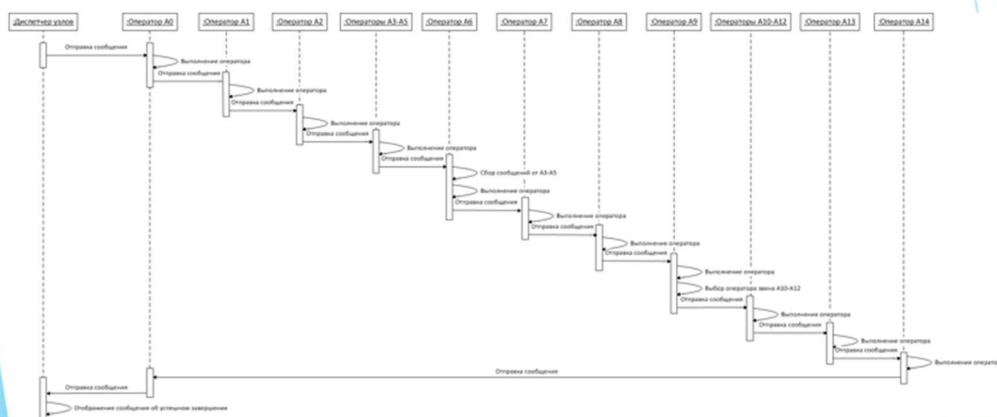
Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090301.29.001 ПЗ

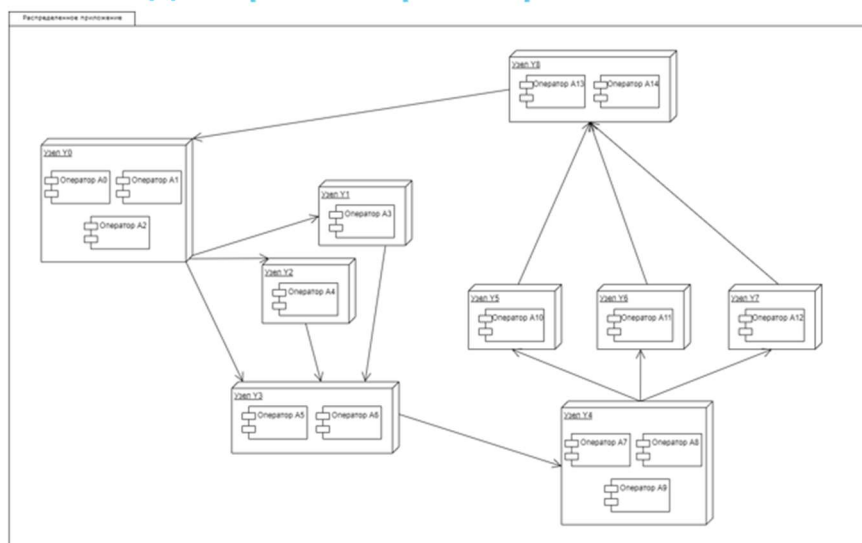
Лист

106

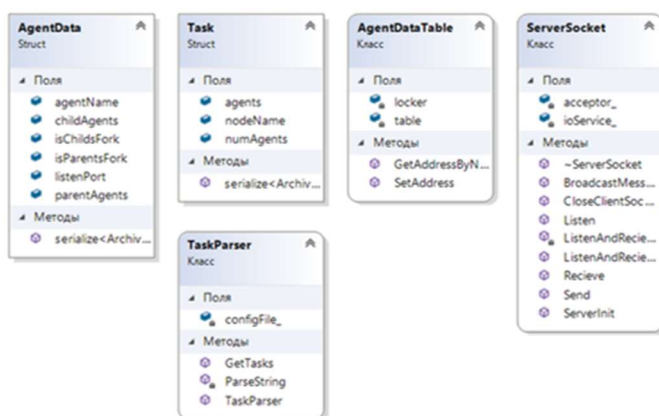
UML - диаграмма последовательностей



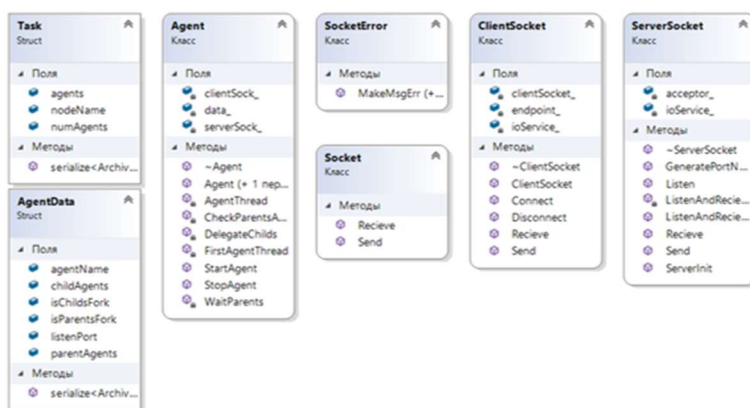
UML - диаграмма развертывания



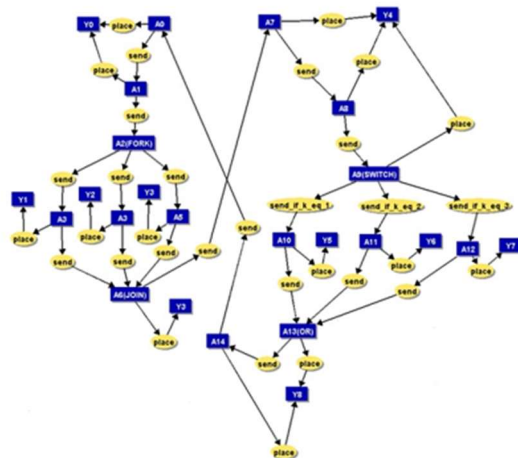
UML - диаграмма классов диспетчера узлов



UML - диаграмма классов удаленного узла



Концептуальный граф



Интерфейс программы диспетчера узлов

```
Сервер запущен
Конфигурационный файл успешно разобран
Прослушивание подключений...
Новый узел подключен - 127.0.0.1:11059 - V0
Новый узел подключен - 127.0.0.1:11061 - V1
Новый узел подключен - 127.0.0.1:11062 - V2
Новый узел подключен - 127.0.0.1:11063 - V3
Новый узел подключен - 127.0.0.1:11064 - V4
Новый узел подключен - 127.0.0.1:11065 - V5
Новый узел подключен - 127.0.0.1:11066 - V6
Новый узел подключен - 127.0.0.1:11067 - V7
Новый узел подключен - 127.0.0.1:11068 - V8
Требуемое число клиентов подключено

Рассылка задач узлам...
Задача успешно выслана узлу! 127.0.0.1:11065
Задача успешно выслана узлу! 127.0.0.1:11062
Задача успешно выслана узлу! 127.0.0.1:11066
Задача успешно выслана узлу! 127.0.0.1:11061
Задача успешно выслана узлу! 127.0.0.1:11063
Задача успешно выслана узлу! 127.0.0.1:11059
Задача успешно выслана узлу! 127.0.0.1:11067
Задача успешно выслана узлу! 127.0.0.1:11068
Задача успешно выслана узлу! 127.0.0.1:11064
Все задачи успешно разосланы узлам

Работа агентов...
Новый агент подключен - 127.0.0.1:11069 - A0
Новый агент подключен - 127.0.0.1:11073 - A1
Новый агент подключен - 127.0.0.1:11071 - A2
Новый агент подключен - 127.0.0.1:11074 - A3
Новый агент подключен - 127.0.0.1:11072 - A4
Новый агент подключен - 127.0.0.1:11075 - A5
Новый агент подключен - 127.0.0.1:11078 - A6
Новый агент подключен - 127.0.0.1:11076 - A7
Новый агент подключен - 127.0.0.1:11081 - A8
Новый агент подключен - 127.0.0.1:11082 - A9
Новый агент подключен - 127.0.0.1:11084 - A10
Новый агент подключен - 127.0.0.1:11080 - A11
Новый агент подключен - 127.0.0.1:11085 - A12
Новый агент подключен - 127.0.0.1:11079 - A13
Новый агент подключен - 127.0.0.1:11086 - A14
Агент A1 завершился успешно
Агент A2 завершился успешно
Агент A3 завершился успешно
Агент A4 завершился успешно
Агент A5 завершился успешно
Агент A6 завершился успешно
Агент A7 завершился успешно
Агент A8 завершился успешно
Агент A9 завершился успешно
Агент A12 завершился успешно
Агент A13 завершился успешно
Агент A14 завершился успешно
Агент A0 завершился успешно
Работа агентов успешно завершена
```

Изм.	Лист	№ докум.	Подпись	Дата

Интерфейс программы удаленного узла

```
Соединение с диспетчером узлов установлено
Ожидание задачи от диспетчера узлов
Задача получена
Узел: Y0
Операторы узла: A0 A1 A2
[ A0 ] Ожидание сигнала о начале вычислений от диспетчера узлов
[ A1 ] Ожидание данных от родительских узлов: A0
[ A2 ] Ожидание данных от родительских узлов: A1
[ A0 ] Сигнал старта получен
Оператор A0 успешно выполнен
[ A0 ] Отправка данных дочерним узлам
[ A0 ] Данные успешно отправлены дочерним узлам: A1
[ A0 ] Ожидание данных от родительских узлов: A14
[ A1 ] Данные получены: A0
Оператор A1 успешно выполнен
[ A1 ] Отправка данных дочерним узлам
[ A1 ] Данные успешно отправлены дочерним узлам: A2
[ A2 ] Данные получены: A1
Оператор A2 успешно выполнен
[ A2 ] Отправка данных дочерним узлам
[ A2 ] Данные успешно отправлены дочерним узлам: A3 A4 A5
[ A0 ] Данные получены: A14
Вычисления завершены успешно.
```

Результаты тестирования на реальной локальной сети

Было проведено 100 запусков программы в условиях реальной вычислительной сети

```
Выполнить: C:\WINDOWS\system32\cmd.exe
Время работы сети при 60 запусках - 3.688682 секунд
Время работы сети при 61 запусках - 3.688135 секунд
Время работы сети при 62 запусках - 3.769188 секунд
Время работы сети при 63 запусках - 3.857180 секунд
Время работы сети при 64 запусках - 3.871273 секунд
Время работы сети при 65 запусках - 3.884292 секунд
Время работы сети при 66 запусках - 2.713378 секунд
Время работы сети при 67 запусках - 3.528337 секунд
Время работы сети при 68 запусках - 2.818711 секунд
Время работы сети при 69 запусках - 3.593142 секунд
Время работы сети при 70 запусках - 3.534584 секунд
Время работы сети при 71 запусках - 2.948885 секунд
Время работы сети при 72 запусках - 2.984773 секунд
Время работы сети при 73 запусках - 2.518278 секунд
Время работы сети при 74 запусках - 2.793271 секунд
Время работы сети при 75 запусках - 3.724942 секунд
Время работы сети при 76 запусках - 3.645432 секунд
Время работы сети при 77 запусках - 3.789274 секунд
Время работы сети при 78 запусках - 3.551512 секунд
Время работы сети при 79 запусках - 3.537878 секунд
Время работы сети при 80 запусках - 2.591254 секунд
Время работы сети при 81 запусках - 3.571357 секунд
Время работы сети при 82 запусках - 3.382588 секунд
Время работы сети при 83 запусках - 3.489378 секунд
Время работы сети при 84 запусках - 3.438564 секунд
Время работы сети при 85 запусках - 3.417757 секунд
Время работы сети при 86 запусках - 3.829633 секунд
Время работы сети при 87 запусках - 3.258183 секунд
Время работы сети при 88 запусках - 2.788892 секунд
Время работы сети при 89 запусках - 3.211951 секунд
Время работы сети при 90 запусках - 3.845681 секунд
Время работы сети при 91 запусках - 2.974229 секунд
Время работы сети при 92 запусках - 3.380112 секунд
Время работы сети при 93 запусках - 2.981866 секунд
Время работы сети при 94 запусках - 3.882195 секунд
Время работы сети при 95 запусках - 2.788482 секунд
Время работы сети при 96 запусках - 3.833126 секунд
Время работы сети при 97 запусках - 3.822848 секунд
Время работы сети при 98 запусках - 3.481389 секунд
Время работы сети при 99 запусках - 3.424549 секунд
Среднее время работы сети при 100 запусках - 3.227716 секунд
Корректное завершение тестов
```

Заключение

- ▶ В ходе выполнения выпускной квалификационной работы была разработана программная реализация сети пиринговых узлов для выполнения распределенных приложений на языке C++. Разработанное приложение позволяет реализовывать взаимодействие удаленных узлов в вычислительной сети и осуществлять решение определенной задачи на конкретном узле.
- ▶ В дальнейшем можно усовершенствовать разработанное приложение и добавить возможность добавления и удаления узлов на схеме во время выполнения вычислений. Добавить механизмы отказоустойчивости, позволяющие продолжать проводить вычисления при отключении одного из узлов.

Спасибо за внимание!

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 1.090301.29.001 ПЗ

Лист

111