

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

**РАЗРАБОТКА РАСПРЕДЕЛЕННОЙ СИСТЕМЫ ИСПОЛЬЗОВАНИЯ УДАЛЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ РЕСУРСОВ**

Работу выполнил В. С. Кузьмин

СОДЕРЖАНИЕ

Введение .................................................................................................................. 3

1. Распределенная вычислительная система ......................................................... 5
   1. Основные понятия и определения ............................................................... 5
   2. Применение ................................................................................................... 7
   3. Примеры распределенных систем ............................................................... 8
      1. BOINC .................................................................................................... 8
      2. MATLAB Distributed Computing Server ............................................ 10
      3. Distributed Computing Technologies Іnc. ............................................ 11
2. Разработка распределенной вычислительной системы ................................. 14
   1. Общие сведения ........................................................................................... 14
   2. Функциональное назначение ..................................................................... 15
   3. Описание логической структуры ............................................................... 17
      1. Связь межу узлами .............................................................................. 17
      2. Основные компоненты ....................................................................... 21
   4. Реализация ................................................................................................... 24
      1. Объектные распределенные системы ............................................... 24
      2. Сравнение CORBA, DCOM и RMI ......................... 26
      3. Состав разработанной программы .................................................... 30
      4. Реализация программы ....................................................................... 33
      5. Тестирование ....................................................................................... 42
   5. Безопасность .............................................................................................. 50

Заключение .......................................................................................................... 52

Список использованных источников .................................................................. 53 Приложение А UML диаграммы классов системы CRDS ................................ 54

Приложение B Исходный код классов Test и Permutation ................................ 57

Приложение С Исходный код классов ClientImpl и ServerManagerImpl ......... 59

ВВЕДЕНИЕ

За последние десятилетия наблюдается невероятно быстрое развитие компьютерных технологий, одной из которых являются распределенные вычисления.

В связи с появлением локальных, а затем глобальных компьютерных сетей, распределенные системы получили большое развитие в теории вычисления и стали внедряться в различные отрасли, использующие ITтехнологии. На сегодняшний день трудно найти систему, которая бы не использовала технологию распределенных вычислений. Распределенные системы повсюду – это компьютеры, умные дома, библиотеки, банки, больницы и т.д.

Такая популярность распределенных систем вызвана самой идеей данной технологии – это объединение в единое целое вычислительные устройства, в роли которых часто выступают персональные компьютеры. Таким образом мы получаем систему, способную работать со всеми её компонентами как с одной целой системой, мощность которой и возможности намного выше.

Распределенные системы позволяют не только организовывать работу с компонентами, через их представление в виде единого элемента, но также осуществлять вычислительные операции, используя процессорное время компьютеров. Данная система, имеющая в своем распоряжении сотни или же тысячи компьютеров, по производительности не уступает суперкомпьютерам. Основным преимуществом распределенной системы – это её низкая стоимость, при высокой производительности.

Известны случаи использования распределенных систем для решения таких сложных по вычислениям задач, как расчёт трёхмерной структуры белков, разработка лекарств, симуляция климатических моделей, вычисление чисел Мерсенна, проверка криптоустойчивости алгоритмов шифрования и многое другое. Большинство таких систем в качестве компонентов используют персональные компьютеры пользователей, которые добровольно предоставляют свои вычислительные ресурсы, что упрощает решение поставленной задачи, а также уменьшает стоимость проекта, так как не требуется приобретение и установка дорогих суперкомпьютеров. Основной идеей приведенных выше систем является использование компьютера во время его бездействия, что повышает эффективность использования устройства. Такая система может быть полезна в крупных компаниях, в офисах которой компьютеры бездействуют длинные промежутки времени. Использование этих промежутков времени для решения важных для компании задач позволит максимально эффективно использовать вычислительную мощность компьютеров как техническом, так и в экономическом плане.

В данной выпускной квалификационной работе целью является разработка системы для распределения вычислительных ресурсов, идея которой состоит в предоставлении программисту возможности распределения своей задачи для вычисления её частей на компьютерах, связанных между собой глобальной (Wide-Area Networks, WAN) или локальной (Local-Area Networks, LAN) сетью. Перед разработкой системы необходимо исследовать технологию распределенных вычислений, а также провести анализ существующих систем подобного рода.

1. Распределенная вычислительная система

1.1 Основные понятия и определения

В области вычислительных систем присутствует раздел, посвященный теоретическим вопросам организации распределенных систем, который именуется как распределенные вычисления.

Теория распределенных вычислений описывает методы разделения решаемой задачи на подзадачи, которые будут выполняться на отдельных вычислительных устройствах. Таким образом, распределённые вычисления чем-то похожи на параллельные, различие состоит в том, что для распределения задачи в качестве вычислительного устройства выступает компьютер, параметры которого должны учитываться, а это объем памяти, производительность процессора, тип соединения (LAN или WAN), задержки

при передаче подзадач, географическое местоположение и т.д.

Распределенные вычисления являются теоретической основой для создания распределенных систем.

У понятия распределенной системы не существует единого определения, так как оно может применяться в разных областях. Для данной дипломной работы введем следующие понятие распределенной системы, которое точно отражает её применение.

Распределенная система – набор связанных между собой компьютеров, которые представлены пользователю единой системой.

В распределенной системе компьютеры никак не зависят друг от друга и являются полностью автономными. Для обеспечения связи между такими компьютерами в системе обычно используется специальное промежуточное программное обеспечение нижнего уровня (рисунок 1), задача которого состоит в сокрытии от пользователя распределённой природы системы, а также в некоторых случаях организовать связь, не зависимую от платформы и операционной системы устройства. На нижнем уровне реализуется основная часть функциональности распределенной системы. Также на данном уровне осуществляется связь с операционной системой устройства и сетевым уровнем. Связь с сетевым уровнем программного обеспечения системы позволяет физически связывать между собой компоненты системы и получать информацию о сетевом соединении, например – используемые протоколы передачи данных, тип связи между устройствами (LAN или WAN) и т.д.

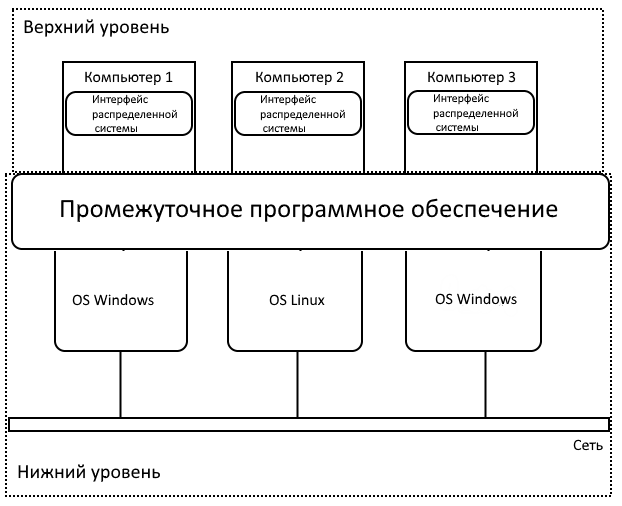


Рисунок 1 – Уровни программного обеспечения распределенной

вычислительной системы

Программное обеспечение верхнего уровня представляет собой интерфейс распределенного приложения для взаимодействия пользователя с системой.

Одним из достоинств распределенных систем является легкое масштабирование, так как для увеличения мощности достаточно подключить к промежуточному слою системы новое устройство.

Помимо предоставления вычислительных ресурсов распределенная вычислительная система способна обеспечить пользователям доступ к общим ресурсам, части которой располагаются на разных устройствах.

Также следует отметить, что система в силу своей распределённой природы, порождает определенные требования к разработчику. Основными требованиями к системе является прозрачность, т.е. скрытие от пользователя всех действий системы организующих распределение; открытость – предоставление служб, имеющих открытый синтаксис и семантику; масштабируемость – способность системы наращивать или сохранять свои характеристики при изменении количества или качества её устройств. Более подробное описание перечисленных требований содержится в источнике[1].

1.2 Применение

Более широкое применение получили распределенные вычисления использующие свои узлы в качестве вычислительных устройств для решения трудных задач моделирования и расчетов. Такие системы предлагают чаще всего пользователям поучаствовать в том или ином проекте, получив тем самым награду в виде результатов совместных исчислениях.

Сама идея использования вычислительных ресурсов компьютеров возникла вовремя, когда персональный компьютер по своим характеристикам приравнялись к суперкомпьютерам 20-30 летней давности. При этом на сегодняшний день большинство пользователей персональных компьютеров используют только 10-40% мощности процессоров, таким образом не расходуется драгоценная часть вычислительных ресурсов, которые могут быть применены для важных задач.

Таким образом по ряду причин были созданы системы, основанные на распределенных вычислениях, позволяющие одним пользователям предоставлять неиспользуемые ресурсы своей системы другим для решения как локальных, так и глобальных задач таких, как поиск лекарств или вычисления в области физики и криптографии.

Также распределенные системы подобного рода позволяют пользователям решить какую-либо свою задачу, требующую огромных вычислений. Стимулом к вычислению пользовательских задач может послужить денежное вознаграждение или интерес участника к проекту, для которого будет задействован его компьютер.

На сегодняшний день существует огромное множество систем, предлагающих пользователям внести свой вклад в ту или иную область, при этом такие системы зачастую используют процессор (центральный или графический) компьютера только во времена простоя компьютера или же с минимальной нагрузкой, обеспечивающей пользователю комфортную работу.

1.3 Примеры распределенных систем

Рассмотрим системы, основанные на распределенных вычислениях, успешность которых доказывает их применимость в области компьютерных вычислений. Данные системы позволяют поучаствовать в крупных или малых проектах, которые могут принести пользу окружающим. Кроме того, как исключение рассмотрено дополнение к системе MATLAB, позволяющие увеличить скорость работы программы за счет использования удаленных компьютеров.

1.3.1 BOINC

Открытая программная платформа Беркли для распределенных вычислений (Berkeley Open Infrastructure for Network Computing, BOINC) – самая популярная некоммерческая платформа для организации распределенных вычислений проекта, где в качестве исполнителей выступают устройства на базе операционных систем Windows, Linux и даже Android. Изначально система разрабатывалась для одного проекта под названием SETI@home, но затем разработчики адаптировали её и для других научно-исследовательских работ.

Данная система, организованная с помощью GRID технологий[2] и клиент-серверной архитектуры, где сервер, в основном, организован на PHP криптах и выполняет действия по управлению рассылкой задач по клиентам и контролем полученных в итоге результатов.

BIONC является именно платформой, так как сами распределенные задачи создаются её пользователями. Каждая задача состоит из таких характеристик по отношению к одному вычислительному узлу, как объем используемой оперативной памяти, память на жестком диске для хранения задачи и результатов вычислений, а также время вычисления одной задачи, которое варьируется от нескольких часов до нескольких дней.

Пользователь данной платформы сам выбирает задачу, в которой он хочет поучаствовать и запускает её на выполнение. Стоит отметить, что данная платформа позволяет одновременно выполнять несколько проектов, если компьютер или смартфон подходит под заявленные требования.

Для того, чтобы добавить задачу на обозрение участникам платформы BOINC, разработчику необходимо создать так называемый boinc-проект, а затем предоставить подробную информацию о своем проекте, включающую: цель, источники данных, системные требования, приложения проекта, особенности приложений, какие именно расчеты будет проводить компьютер, результаты работы проекта и другая дополнительная информация. Вся перечисленная информация устанавливается разработчиком на сайте проекта BOINC[3], на котором содержится вся информация о действующих и завершенных проектах.

Таким образом задача становится открытой для пользователя и тот, если хочет внести свой вклад в проект и его компьютер или смартфон удовлетворяют требованиям, запускает её на выполнение, за что в итоге получает балы и возможность наблюдать процесс решения задачи.

За своё существование платформа BIONC решила 22 крупных задачи, например, проект Asteroids@home – подтвердил эффективность используемого метода поиска и анализа астероидов большого и среднего диаметра, построив 328 моделей астероидов и выявив особенности эффекта Ярковского для астероидов, пролетающих в пределах Земли. На данный момент в проекте содержатся порядка 50 задач из таких областей как медицинская биология, естественнонаучная, математическая и прочие.

В проектах такого рода участвуют люди со всего мира, которые чаще всего объединяются в команды, которые соревнуются между собой по количеству выполненных вычислений, что придает проекту больше вычислительных ресурсов. Таким образом платформа BOINC объединила компьютеры и смартфоны со всего мира для решения задач, которые могут принести пользу как науке, так и человечеству. Пожалуй, это самая крупная распределённая вычислительная сеть на данный момент.

1.3.2 MATLAB Distributed Computing Server

MATLAB Distributed Computing Server – это пакет для MATLAB обеспечивающий организацию распределенных вычислений в данной среде.

В качестве клиентской части используется продукт Parallel Computing Toolbox.

Parallel Computing Toolbox представляет собой инструмент использующий для вычисления задач MATLAB параллельные вычисления с помощью центрального или графического процессора.

В процессе распределения задач между исполнителями в MATLAB участвует планировщик Math Works Job Manager, который также координирует выполнением заданий.

В MATLAB Distributed Computing Server также имеется возможность интегрировать средства для распределенных вычислений других разработчиков, благодаря поддержке типовых интерфейсов.

Описанные инструменты позволяют автоматически превратить систему MATLAB в распределенную систему, которая способна сама распределить задачу. Подобная технология выделяет данные инструменты среди других систем. Однако, минусом является, то что изначально нет доступной сети устройств, готовых выполнить задачу, вместо этого разработчик сам занимается поиском и привлечением участников к своему проекту.

Так как MATLAB является популярной платформой для ведения различного рода вычислений в научных областях, то порой возникают ситуации, требующие огромных вычислительных ресурсов, которые могут обеспечить несколько устройств, соединённые между собой сетью и использующие выше описанные инструменты.

Данная распределенная система не носит публичный характер, в отличии от остальных рассмотренных проектов, однако позволяет создавать свою собственную систему для решения задачи, позволяющей проводить научные исследования и тем самым приносить пользу.

1.3.3 Distributed Computing Technologies Іnc.

Distributed Computing Technologies Іnc. или distributed.net – самая старая из приведенных распределенных систем, но тем не менее одна из популярных. Данный проект основан 1997 и по сей день работает над распределенными вычислениями в двух проектах RC5-72 и ORG. Суть

проекта заключается в решении математических и криптографических задач, для решения которых не существует оптимальных алгоритмов, таким образом вычисления выполняются методом полного перебора.

Проект RC5-72 представляет собой демонстрацию уязвимости шифра RC5 с использованием 72 битного ключа, путем поиска ключа полным перебором, в котором участвуют тысячи компьютеров пользователей, состоящих в командах. Стимулом решения данной задачи является денежное вознаграждение команде, нашедшей ключ, от конкурса Secret Key Challenge проводимое компанией RSA Labs.

До этого проекта, distributed.net решили следующие криптографические задачи:

а) 1997 году за 250 дней был найден 56-битный ключ RC5-32/12/7;

б) 1998 году за 40 дней был найден 56-битный ключ DES-II-1;

в) 1999 году с помощью системы DES “Deep Crack” был найден ключ

алгоритма DES-III;

г) 2000 году за 62 дня был найден ключ алгоритма CSC;

д) 2002 году за 1757 дней был найден 64-битный ключ алгоритма RC5. Вторым проектом, над которым работает distributed.net, является ORG (Optimal Golomb Rulers) – математический проект, целью которого стоит поиск оптимальных линеек Голомба. Решение данной задачи важно для теории связи, радиоастрономии и рентгено-кристаллографии.

За все время существования проекта были доказаны оптимальности линеек 24 (в 2004 году), 25 (в 2008 году), 26 (в 2009 году) и 27 (в 2014 году) порядка.

Вычисление в описанной системе осуществляются с помощью клиентов, связанных между собой локальной или глобальной сетью. Клиенты синхронизируются между собой и передают данные о выполненных вычислениях. Сервер в данном случае выступает в качестве раздающего задачи узла и принимающего промежуточные результаты. Помимо обычных клиентов в системе присутствует персональный прокси, задача которого контролировать и раздавать задачи устройствам своей подсети, часто не имеющим доступ в интернет и являющимся локальными. Таким образом клиенты подсети выполняют поставленные задачи, синхронизируются между собой и передают информацию персональному прокси, который обрабатывает полученные данные и отправляет другим участникам сети, а также серверу.

Программа клиента и персонального прокси имеет малый размер и реализована для более 20-ти операционных систем, среди которых присутствуют Windows, Linux и Mac.

Задача, поступившая клиенту, занимает в дисковом пространстве всего несколько сотен байт и выполняется от нескольких минут до 2-3 часов.

Из описанных выше распределенных систем, представляющих собой множество объединенных компьютеров обеспечивающих поставленной задаче вычислительные ресурсы сравнимые с ресурсами суперкомпьютеров или же превышающие их следует, что системы подобного рода существуют и являются эффективными.

Таким образом необходимость в таких системах состоит в том, что для вычисления некоторых задач нет возможности предоставить суперкомпьютер, что является проблемой недостающих ресурсов производительности, решением которой становится использование системы, распределяющей задачу по устройствам, ресурсы которых применяются для вычислений. Распределенные системы демонстрируют полный потенциал, применения лишних вычислительных ресурсов персональных компьютеров.

Также важным в таких системах является найти пользователя, которому не безразлична ваша задача. Из описанных систем следует, что в мире существует много людей, которые готовы помочь в вычислениях с помощью своего компьютера. Это доказывает необходимость разработки систем подобного рода.

1. Разработка распределенной вычислительной системы

2.1 Общие сведения

В ходе дипломной работы, целью которой является написание системы, позволяющей распределить вычислительные ресурсы, была разработана программа CRDS (Computational Resource Distribution System) на языке программирования Java.

Так как система написана на языке Java, то для её работы необходима виртуальная машина Java (Virtual Machine Java, JVM).

Программа CRDS работает с операционными системами, поддерживающими JVM, к которым относятся системы семейства Windows, Linux и Mac OS X.

Для работы системы желателен доступ в интернет или в локальную сеть.

В состав системы CRDS входят:

а) CRDS.jar – архив для JVM, который содержит все необходимые для работы системы компоненты, а также классы ComputeEngine, CRDSController, ServerStatus, TaskStatus и интерфейс Task, доступные пользователю.

б) client\_policy, crds\_policy, server\_policy – файлы с конфигурациями

системы безопасности Java;

в) json-simple-1.1.1.jar – библиотека для работы с файлами формата

JSON;

г) lipermi-0.4.jar – сторонняя библиотека LipeRMI[4] для работы с

технологией RMI.

Так как система написана с применением JDK 8, то использование более ранних версий может привести к ошибкам в работе программе.

Для работы системы CRDS в глобальной сети интернет используются специальные веб серверы, для общения с которыми используются POST запросы, содержащие файлы формата JSON. Веб сервер состоит из базы данных и PHP скрипта для обращения к этой базе данных.

2.2 Функциональное назначение

Данная программа представляет собой систему, которая осуществляет связь между компьютерами и с помощью библиотек, входящих в её состав, предлагает пользователю на высоком уровне абстракции распределить свою задачу, а затем получить результат.

В состав программы входят классы ComputeEngine и CRDSController, которые доступны пользователю и содержаться в файле CRDS.jar.

Для установки программы на компьютер достаточно скопировать файл CRDS.jar, папку с библиотеками lib и файлы crds\_policy, client\_policy, server\_policy.

Класс ComputeEngine представляет собой реализацию абстрактного вычислителя, который на вход принимает задачу, а на выходе выдает результат. Задача определяется пользователем через класс, реализующий интерфейс Task, в качестве результата выполнения задачи, класс ComputeEngine возвращает пользователю объект класса реализации Task, таким образом пользователь сам решает какую структуру будет иметь его задача и как будет представлен результат, который он может получить определенным методом. Класс ComputeEngine при получении задачи взаимодействует с запущенной программой CRDS, которая осуществляет поиск компьютера с установленной системой CRDS, куда и отправляется поставленная задача. Компьютеры могут находится как в локальной, так и в глобальной сети. Таким образом использование класса ComputeEngine и системы CRDS обеспечивать прозрачность взаимодействия пользователя с удаленными вычислительными машинами.

Класс CRDSController обеспечивает пользователю контроль над системой CRDS, а именно возможность изменять максимальное количество отправляемых задач, принимаемых задач, настройка количества используемой памяти и другие. В добавок присутствуют методы контроля состояний задач, отправляемых и выполняемых системой, которые представлены в виде классов перечислений (enum) ServerStatus и TaskStatus. Также при применении связи с устройствами в глобальной сети используются веб сервера, состав которых может быть изменен пользователем через класс CRDSController.

Стоит отметить, что пользователь сам реализует распределенный алгоритм с помощью класса ComputeEngine, система CRDS выступает в роли распределителя как поступающих, так и отправленных задач между устройствами сети.

Также система CRDS ориентирована на пользователей, обладающих базовыми навыками программирования на языке Java.

Помимо решения дорогостоящих, по требуемым вычислительным ресурсам, задач, данная система позволяет решать и другие проблемы, в рамках которых требуется вычисления на других устройствах.

Для запуска системы требуется установленная на компьютере среда выполнения программ Java – это JVM. Также требуется разрешить доступ приложению к сети, путем подтверждения всплывающего сообщения при первом запуске программы или вручную добавить в исключения брандмауэра. Также при использовании для подключения к сети через роутер, применяемый NAT, следует перенаправить данные программы в настройках роутера.

В противном случае система, находящаяся за NAT или блокированная брандмауэром, не сможет обеспечить пользователю полный набор функций, а именно – работа компьютера в качестве узла для других устройств сети.

Таким образом для выделения сервера со стороны системы, чтобы обеспечить доступ через интернет требуется белый IPv4 адресс[5] и правильно настроенный брандмауэр, а также роутер (если используется). Однако, в отсутствии предъявленных требований, программа может функционировать только в пределах компьютера пользователя.

2.3 Описание логической структуры

2.3.1 Связь межу узлами

Рассмотрим общую концепцию работы программы CRDS, как распределенной системы.

Распределенная система, построенная на узлах, связь с которыми обеспечивает программа CRDS представляет собой децентрализованную структуру.

Децентрализованная распределенная система – это система, в которой компоненты для связи не используют единый центр – сервер. Таким образом, каждый узел системы представляет собой как сервер, так и клиент. Связь между такими узлами обеспечивается соединением p2p (peer-to-peer)[2].

Преимуществом таких систем, по сравнению с централизованными, где главным узлом является сервер, является устойчивость к отказам, т.е. отказ одного из узлов не влияет на работоспособность всей системы. Единственным недостатком является безопасность данных передаваемых в такой сети, однако, при определенной архитектуры этот недостаток можно устранить.

Рассмотрим структуру и способы взаимодействия в распределенной системы на базе приложения CDRS.

Покажем метод установки связи системы на примере установки соединения между двумя компьютерами, которые назовем компьютер 1 и компьютер 2 (Рисунок 2). На обоих устройства установлено приложение CRDS, которая при манипуляциях программы пользователя на компьютере 1 инициирует связь с компьютером 2. Также для установки связи в качестве информаторов выступают веб сервера, пусть в рассматриваемом примере присутствуют n серверов, пронумерованных от 1 до n.

Установка связи между компьютерами 1 и 2 происходит в 2 этапа, при этом используем переменную i (1 ≤ i ≤n), которая будет индексом веб сервера и изначально равняется 1:

1. CRDS обращается к веб серверу i который в ответ на запрос отправляет информацию по другим устройствам зарегистрированных на данном веб сервере.
2. Система принимает ответ и на основе полученной информации пытается подключится к одному из устройств, в случае успеха устанавливается связь между устройством (в нашем случае это компьютер 2). Если же не удалось подключится ни к одному из устройств или присланный ответ не содержал информации об устройствах, то i=i+1 и перейдем к шагу

(1).

Данные действия производятся до тех пор, пока не будет установлена связь с другим устройством или же при опросе всех серверов не будет найдено ни одного устройства.

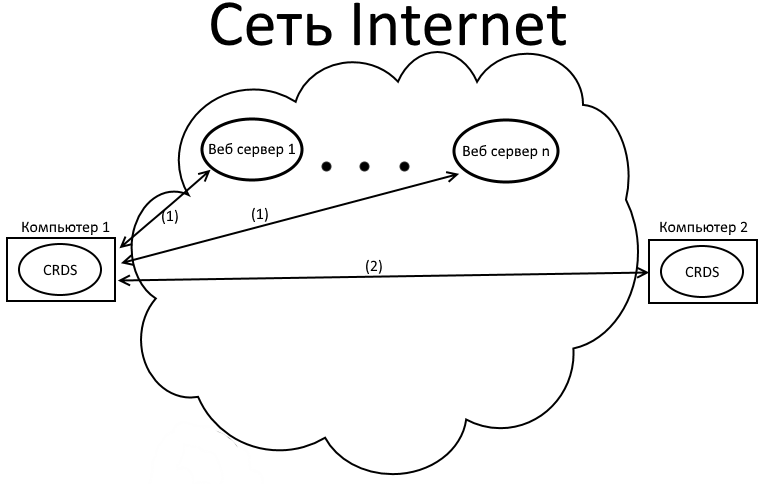


Рисунок 2 – Установка глобальной связи между двумя компьютерами с

помощью CRDS

Следует отметить то, что CRDS регистрирует компьютер пользователя во всех серверах доступных во время запуска данного приложения.

Помимо глобальной сети CRDS поддерживает установку связи по локальной сети. Рассмотрим пример, демонстрирующий установку связи в локальной сети (Рисунок 3).

Пусть в нашем распоряжении имеются n компьютеров, которые связанны между собой локальной связью и имеют установленный и запущенный CRDS. Тогда для того чтобы установить связь между компьютером 1 и компьютером i, где 2 ≤ i ≤n, требуется выполнить следующие действия:

1. CRDS на компьютере 1 выполняет широковещательную передачу параметров, которые должен предоставлять CRDS на другом устройстве.
2. Компьютеры с индексами от 2 до n принимают переданные параметры, сравнивают с параметрами своей системы CRDS и отправляют ответ от CRDS компьютера 1. В качестве ответа выступают параметры для подключения, если система удовлетворяет требованиям, иначе отправляется отказ.

Компьютер 1 устанавливает связь с тем устройством, которое первым прислало ответ с разрешением на подключение, остальные ответы игнорируются.

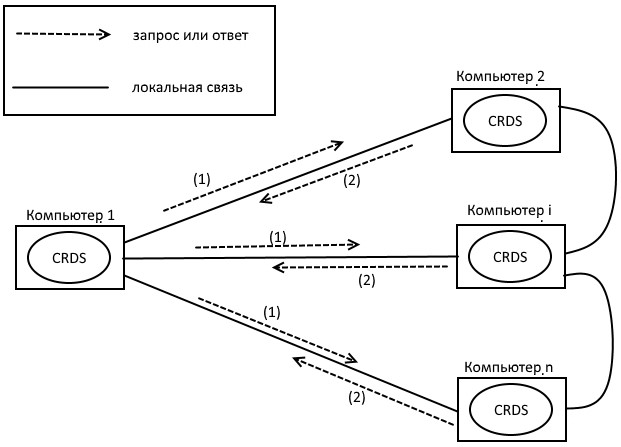


Рисунок 3 – Установка локальной связи между компьютерами с

помощью CRDS

Таким образом распределенная система, построенная на применении приложения CRDS, обеспечивает связь устройств как в локальной сети, так и в глобальной. Следует отметить, что глобальное соединение в CRDS находится в более высоком приоритете, поэтому первоначально осуществляется поиск устройства в глобальной сеть, а уж затем, если не удалось найти веб сервер или свободный сервер на компьютере, в локальной сети. Более подробная организация будет рассмотрена далее.

2.3.2 Основные компоненты

Сформировав общую картину взаимодействия узлов системы, рассмотрим структуру самого узла, а именно программы CRDS позволяющей компьютеру выполнять роль компонента распределенной системы.

Система CRDS состоит из следующих компонентов, определяющих основную часть функционала:

1. CRDS – сама система, организующая и управляющая работой остальных компонентов.
2. Client или клиент – основной компонент отвечающий за клиентскую часть работы системы CRDS, именно этот компонент отправляет запросы на связь, а также обрабатывает ответы, на основании которых принимает решения по подключению. Также данный клиент является интерфейсом для взаимодействия пользователя с системой.
3. ServerManager или менеджер серверов – основной компонент отвечающий за отправку ответа на запрос Client, а также дальнейшей установки связи и выделения для клиента вычислительной единицы, т.е. сервера. Информация о выделенных серверах хранится в менеджере серверов на основании которой ServerManager принимает решения в тех или иных ситуациях.
4. ComputeEngine или вычислитель – управляющая компонента, которая доступна пользователю в абстрактном виде и по действиям которого, инициирует запрос от Client на новое подключение. После установки связи вычислитель работает напрямую с сервером, выделенным менеджером серверов.
5. CRDSController или контроллер CRDS – управляющая компонента, позволяющая пользователю контролировать параметры системы CRDS.

Рассмотрим работу описанных выше компонентов при установке связи между двумя компьютерами (Рисунок 4).

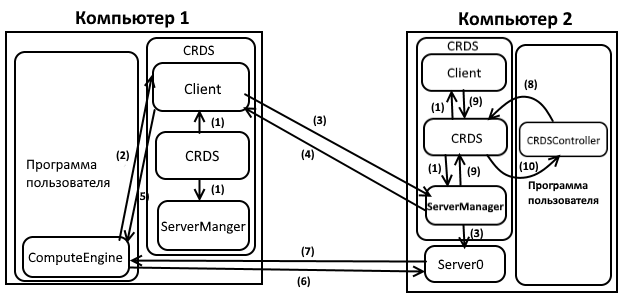


Рисунок 4 – Взаимодействие компонентов CRDS между двумя компьютерами

Пусть в нашем распоряжении два компьютера с номерами 1 и 2. Во время запуска приложения CRDS, изначально создается компонент CRDS, который формирует Client и ServerManager, устанавливает связь (1) и продолжает работу в фоновом режиме.

При запуске пользователем своей программы на компьютере 1, которая использует компонент ComputeEngine для выполнения поставленной задачи, вычислитель отправляет запрос клиенту (2) на предоставление удаленного сервера. Client связывается описанными выше методами с ServerManager (3) CRDS компьютера 2, через локальную или глобальную сеть. ServerManager принимает запрос от клиента, загружает программу задачи, затем резервирует и запускает Server0 (3) на основе параметров полученных от клиента. После чего менеджер серверов отправляет данные клиенту для доступа к Server0 (4). Client пересылает полученные данные ComputeEngine (5), после чего тот устанавливает соединение с Server0 (6), передаёт ему задачу на выполнение и через некоторое время, запросив результат получает

(7).

Также на рисунке 4 рассмотрен случай применения CRDSController на компьютере 2 для управления CRDS. Пользователь пишет свою программу, которая с помощью компонента CRDSController может изменять параметры CRDS или её основных компонентов. При запуске программы CRDSController обращается к компоненте CRDS (8) с целью изменить параметры или получить данные. Система в ответ на полученный запрос изменяет (при необходимости) параметры сначала в CRDS, а затем в её компонентах (9), после чего возвращает CRDSController результат (10),

который пользователь может использовать в своих целях.

Использование описанной структуры CRDS уменьшает нагрузку на главные компоненты Client и ServerManager, так как после установки связи между ComputeEngine и удаленным сервером Server0, действия производятся только через эти компоненты, не затрагивая главные, которые в это время могут обрабатывать другие запросы.

Сформируем основные требования и свойства описанных компонент.

Client – выполняется под контролем CRDS в одном процессе, но в другом потоке. Клиент должен обеспечивать должен обеспечивать одновременный доступ к своему функционалу сразу нескольким объектам ComputeEngine. Также Client в процессе формирования связи должен быть устойчивым к сбоям. В случае, когда компьютер пользователя не подключен локальной или глобальной сети, клиент должен создать связь с локальным компонентом ServerManager и предоставить сервер от текущего компьютера, что гарантирует пользователю, что его задача будет выполнена. Клиент должен адаптироваться под настройки, внедряемые CRDS в процессе работы программы. Действия клиента полностью зависят от типа сети.

ServerManager – выполняется под контролем CRDS в одном процессе, но в другом потоке. Данный компонент должен обеспечивать доступ к своему функционалу сразу нескольким объектам Client. ServerManagar должен быть устойчивым к сбоям и не зависеть от типа используемых подключений. Главная задача компонента – это управление запусками и остановкой серверов, которые включают в себя: установка максимально доступной памяти сервера, считывание информации, выдающиеся сервером в процессе работы – это выходные потоки и потоки ошибок, удаление процесса сервера при достижении им определенного статуса, загрузка файлов для выполнения задачи и запуск сервера, выделение и удаление директории для серверных фалов. ServerManager должен адаптироваться под настройки внедряемые CRDS в процессе работы программы.

CRDS – выполняется в отдельном потоке в одном процессе с Client и ServerManager. Как главный компонент, CRDS должен обеспечить работоспособность зависимых компонентов, а также осуществлять внедрение настроек, вводимых пользователем. CRDS доступен только в пределах компьютера (при использовании ip адреса 127.0.0.1) и только через компонент CRDSController.

ComputeEngine – выполняется в процессе работы пользовательской программы, зависит только от Client и сервера. Должен предоставлять пользователю возможности установки задачи, получение результатов синхронно и асинхронно. Работает вычислитель только при запущенном CRDS. Количество устанавливаемых задач равна одному, для дополнительных задач необходимо создать новый объект класса ComputeEngine. Важной задачей является обеспечить гарантию

выполнимости поставленной задачи, за исключением случаев обнаружения ошибок в программном коде пользователя или возникновением сбоя выполняемой среды.

CRDSController – выполняется в процессе работы пользовательской программы, зависит только от компоненты CRDS. Должен выполнять команды пользователя для изменения или получения состояния системы CRDS. При ошибках со стороны пользователя или CRDS должен

проинформировать пользователя и не допустить введения изменений в систему.

2.4 Реализация

2.4.1 Объектные распределенные системы

Для реализации программы было принято решение использовать технологию разработки распределенных систем объектов.

В основе объектных распределенных систем лежит представление служб и ресурсов системы в качестве удаленных объектов.

Удаленный объект представляет собой набор методов и данных, которые образуют состояние объекта, что позволяет изменять состояния вызовом методов объекта.

Использование объектов в распределенной вычислительной системы позволяет автоматически обеспечить прозрачность системы, так как свойство инкапсуляции объектов позволяет скрыть распределенную структуру системы.

Обычно, доступ к объектам системы осуществляется через специальные, строго определенные интерфейсы. Вызов методов, описанных в интерфейсе, осуществляется удаленно между компонентами системы.

Если возникает необходимость передать объект из одного узла системы в другой, то осуществляется механизм сериализации объекта, путем перевода состояния объекта в последовательность битов, которые передаются принимающему узлу, а затем десериализуются механизмом обратным сериализации.

Объектные распределенные системы имеют ряд преимуществ, по сравнению со стандартной реализацией системы на основе клиент-серверной архитектуры, протоколы взаимодействия в которой необходимо реализовывать самостоятельно. В объектных системах взаимодействие между компонентами определенно соответствующей архитектурой. Таким образом для разработчика упрощается реализация распределенной системы.

Ещё одним немаловажным преимуществом, при использовании объектных распределенных систем, является кроссплатформенность, которая первоначально обеспечивается архитектурой выбранной технологии.

Следует также отметить, то что работать с распределенной системой объектов достаточно легко и быстро, а уровень абстракции системы, которую предоставляет использование объектов, позволяет разрабатывать более сложные и эффективные системы.

Единственной проблемой при использовании удаленных объектов в распределенной системе является время жизни, а именно, момент, когда объект был создан и длительность его существования.

2.4.2 Сравнение CORBA, DCOM и RMI

Для выбора необходимой технологии разработки объектных распределенных систем рассмотрим следующие наиболее популярные распределенные системы – это CORBA, DCOM и RMI.

CORBA (Common Object Request Broker Architecture) или обобщенная архитектура брокера объектных запросов – это технология разработки распределенных приложений, обладающих свойством кроссплатформенности.

Разработкой CORBA занималась группа управления объектами (Object Management Group, OMG), целью которой стояло разработка и продвижение объектно-ориентированных технологий и стандартов для создания корпоративных платформо-независимых распределенных приложений.

Принцип работы CORBA заключается в описании специального интерфейса на IDL (Interface Definition Language) для объекта, находящегося на сервере, после чего для клиента генерируется заглушка, которая также описывается на IDL. С помощью заглушки клиент может вызвать методы объекта, находящегося на сервере, тем самым изменяя его состояние. Интерфейс описывающий методы объекта на сервере называется скелетоном. Скелетон и заглушка позволяют выполнять запросы и ответы между сервером и клиентом, при этом выполняя маршалинг и демаршалинг данных.

В CORBA также присутствуют динамические версии скелетона и заглушки – это DII (Dynamic Invocation Interface) и DSI (Dynamic Skeleton Inerface), позволяющие динамически обнаруживать и вызывать методы объекта. Передача данных между клиентом и сервером осуществляется с помощью брокера объектных запросов (ORB). Применение IDL для описания интерфейсов позволяет поддерживать программы, написанные на таких языках программирования, как C/C++, Java, Ada, COBOL, Python и другие.

Что осуществляется перекомпилированием интерфейса IDL на любой из описанных языков с помощью соответствующего компилятора.

Таким образом преимуществом технологии CORBA является:

* независимость от операционной системы;
* независимость от языка программирования;
* динамические вызовы и обнаружения объектов;
* масштабируемость;
* наличие дополнительных сервисов, упрощающих разработку; - широкая индустриальная поддержка.

Недостатки CORBA следующие:

* отсутствие наименования через URL;
* отсутствие передачи параметров по значению;
* отсутствие динамической загрузки компонентов-переходников.

DCOM (Distributed Component Object Model) - технология для разработки распределенных систем разработанное компанией Microsoft и является расширением технологии COM [6].

Создание объекта в DCOM осуществляется путем вызова менеджера управления сервисами (Service Control Manager) клиента, после чего тот связывается с SCM сервера, после чего передает запрос на создание объекта.

Как и в CORBA в DCOM существует возможность вызова методов удаленного объекта, для этого также предварительно создаются заглушка (стаб) для сервера, содержащий объект, и прокси для клиента, данные интерфейсы описываются с помощью IDL, которые компилируются специальным компилятором в исходный код языка Си.

При вызове методов удаленного объекта с использованием прокси, указанные параметры извлекаются из стека, затем перемещаются в буфер (маршалинг) и передаются через сеть серверу на стаб, через который производится воссоздание стека (демаршалинг), после чего вызывается метод объекта. Затем по завершению работы методы над выходным значением и параметрами выполняется маршалинг, а затем пересылается клиенту. Процесс маршалинга является нетривиальным, так как в качестве параметров могут применяться указатели на массивы и структуры, которые тоже могут содержать указатели.

Преимущества DCOM:

* независимость от языков программирования;
* динамический вызов и поиск объектов; - масштабируемость.

Недостатки:

* сложность реализации;
* зависимость от платформы Windows;
* отсутствие именования через URL;
* отсутствие контроля безопасности на уровне выполнения ActiveX компонент.

RMI (Remote Method Invocation) – технология разработанная компанией JavaSoft для разработки распределенных систем на языке Java и внедренный в JDK с версии 1.1.

Для выполнения вызова методов объекта на стороне сервера необходимо описать интерфейс на Java, который реализуется классом объекта. После создания экземпляра объекта, его необходимо привязать к RMI регистру, указав наименование в виде URL. Определенный интерфейс называется скелетоном и является переходником для сервера. Для клиента генерируется переходник (заглушка) в виде интерфейса, через который осуществляется обращение к RMI регистру и получение ссылки на удаленный объект. Таким образом RMI регистр используется как хранилище ссылок на удаленный объект. Заглушка и скелетон компилируется автоматически, также заглушка может быть передана клиенту динамически.

Достоинства RMI:

* быстрая и простая разработка;
* встроенная безопасность (наличие SecurityManager);
* динамическая загрузка компонент-переходников;
* возможность передачи объектов в качестве параметра;
* Java оптимизация;
* независимость от операционной системы.

Недостатки:

* поддержка только языка Java;
* отсутствие поддержки некоторых общих протоколов взаимодействия; - трудность интегрирования с существующими приложениями.

В приведенных выше технологиях RMI и CORBA разработки

распределенных систем общим достоинством является поддержка различных операционных систем. При этом CORBA и DCOM обладают огромным функционалом и поддерживают множество языков программирования, в отличии от RMI. Однако, самым важным параметром для разработки системы поставленной целью данной дипломной работы являются простота разработки, применение ссылок URL и возможность передачи объектов, как параметров, в чем выигрывает технология RMI. Более подробное сравнение описание рассмотренных технологий присутствует в источнике[8].

Таким образом описанная выше система разрабатывается только для распределения задач, написанных на языке Java. И несмотря на то, что RMI

поддерживает только один язык, данный язык является наиболее популярным среди программистов и позволяет разработать распределенную систему достаточно уникальную и эффективную.

В разработанной программе все основные её компоненты реализованы с помощью технологии RMI по инструкции официального разработчика[7]. Однако, для возможности дальнейшего улучшения программы, а именно поддержки смартфонов с операционной системой Android, на которой не поддерживается RMI от разработчиков Java, было принято решение некоторые компоненты реализовать с применением LipeRMI – библиотека, реализующая технологию RMI с поддержкой Android.

2.4.3 Состав разработанной программы

Система CRDS, разработанная на языке Java с применением технологии RMI в соответствии с описанными моделями связи и взаимодействия, включает в себя следующие классы и интерфейсы: а) Пакет client.

1. Client.java – интерфейс RMI, используемый для связи с клиентом системы CRDS.
2. ClientImpl.java – реализация интерфейса Client, также является реализацией компонента клиента системы CRDS. Экземпляр является объектом RMI.
3. ClientThread.java – класс, унаследованный от класса Thread, создёт, регистрирует, а затем запускает объект класса ClientImpl.

Объект класса ClientImpl запускается системой CRDS, как поток.

1. ConnectThread.java – класс, унаследованный от класса Thread, представляет собой поток, обслуживающий запросы пользователей, т.е. выполняет поиск и соединение с ServerManager. Запускается в нескольких экземплярах, как поток в ClientImpl. б) Пакет compute\_engine.
2. ComputeEngine.java – класс реализующий управляющую компоненту системы CRDS, объект которого позволяет пользователю передать задачу на выполнение удаленной машине и получить результат. Открыт для пользователя.
3. ClientConnectorThread.java – класс, унаследованный от класса Thread, обращается к клиенту за получением ссылки на сервер, устанавливает связь с сервером и передаёт задачу. Запускается как поток в классе ComputeEngine. в) Пакет connector.
4. Connector.java – интерфейс, который содержит функции получения ссылки на менеджера серверов его регистрации и удаления в глобальной и локальной сети.
5. LocalConnector.java – класс реализующий интерфейс Connector для работы с локальной сетью.
6. LocalRecieverThread.java – класс, унаследованный от Thread, выполняет прием запросов и отправку ответов класса LocalConnector другого устройства.
7. WebConnector.java – класс реализующий интерфейс Connector для работы с глобальной сетью и веб сервером.
8. WebRequest.java – класс для формирования запросов веб серверу.

г) Пакет crds.

1. CRDS.java – класс реализации основного компонента системы CRDS, отвечающего за работу остальных компонентов. Хранит данные о настройке системы и осуществляет изменение параметров других компонентов. Экземпляр является объектом RMI. Реализует интерфейс PublicCRDS.
2. PublicCRDS.java – интерфейс RMI, описывающий методы контроля и управления объекта класса CRDS.
3. CRDSController.java – класс, позволяющий управлять и контролировать систему CRDS. Доступен для пользователя.
4. ScheduledTask.java – класс, являющийся задачей для класса java.util.Timer, осуществляет проверку состояний запущенных серверов и при необходимости инициирует запрос менеджеру серверов на закрытие сервера. Запускается в CRDS.java. д) Пакет server.
5. IServer.java – интерфейс RMI описывающий методы сервера.
6. IServerManager.java – интерфейс RMI описывающий методы менеджера серверов.
7. ProcessController.java – класс предоставляющий возможность запуска сервера по заданным параметрам как отдельного процесса, а также осуществляет перехват потоков ошибок и выходных данных сервера.
8. ServerImpl.java – класс реализации сервера, является объектом RMI и обеспечивает выполнение поставленной удаленной машиной задачи, а также возвращает результат вычислений.
9. ServerInfo.java – класс, описывающий параметры сервера.
10. ServerManagerImpl.java – класс реализации менеджера серверов, осуществляет контроль серверов и получение запросов от клиента. Экземпляр является объектом RMI.
11. ServerManagerThread.java – класс, унаследованный от класса Thread, реализует создание объекта ServerManagerImpl и выполняет регистрацию объекта в локальной и глобальной сети.
12. ServerStatus.java – класс перечислений, содержит возможные статусы сервера.
13. ServerThread.java – класс, унаследованный от класса Thread, выполняет задачу, установленную удаленным пользователем.

Вызывается как поток в класск ServerImpl. е) Пакет task.

1. Task.java – интерфейс задачи, описывающий один метод execute(), который реализуется пользователем.
2. TaskStatus.java – класс перечислений, содержит возможные статусы задачи до её передачи серверу.

ж) Пакет zip.

1) ZipArchivator.java – класс, реализующий методы упаковки и распаковки данных архивом Zip.

Структура выше описанных классов и интерфейсов описана в виде

UML диаграмм классов, которые представлены в ПРИЛОЖЕНИЕ A.

2.4.4 Реализация программы

Рассмотрим с помощью UML диаграмм последовательностей общий механизм взаимодействия компонентов системы CRDS.

Итак, сначала представим общую схему взаимодействующих между собой компонент, для осуществления соединения и передачи задачи, написанной пользователем, с одного компьютера на другой. Данная схема была представлена на рисунках 3-4 в разделе 2.2. Теперь, основываясь на схеме рисунка 4, рассмотрим взаимодействия между группами компонентов.

Сперва, рассмотрим подробно взаимодействие объекта класса ComputeEngine с ClientImpl и ServerImpl, при этом опустим тонкости работы ClientImpl и ServerManager (Рисунок 5).

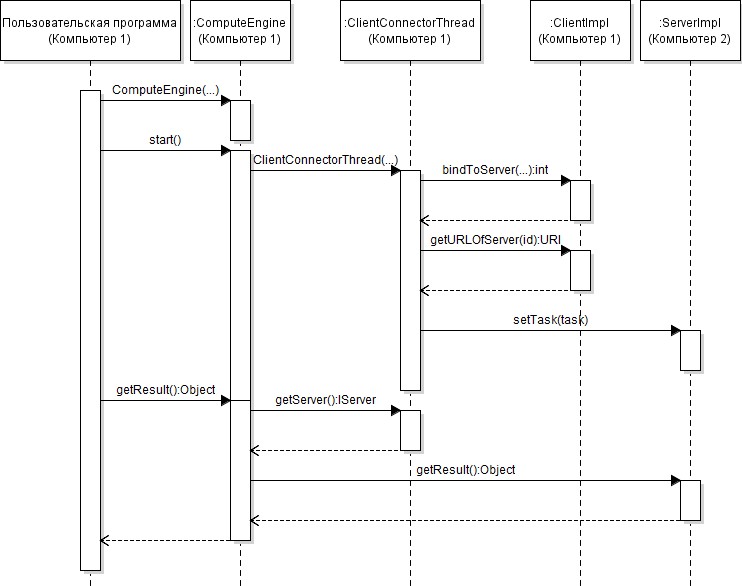


Рисунок 5 – Взаимодействие ComputeEngine с ClientImpl и ServerImpl

При запуске пользователем программы, которая создает экземпляр класса ComputeEngine, при указании в качестве параметров объект класса, реализующего интерфейс Task, объект ComputeEngine получает у RMI регистра ссылку на объект ClientImpl.

Затем, когда пользовательская программа активирует вычислитель методом start(), в объекте создается экземпляр класса ClientConnectorThread и запускает его как поток, конструктору которого передаются ссылка на ClientImpl, объект задачи, путь к файлам реализации задачи и количество памяти необходимое для выполнения. ClientConnectorThread выполняет запрос к ClientImpl - bindToServer(…), у которого в качестве параметров является путь к файлу реализации задачи и количество необходимой памяти, в ответ от ClientImpl получает индекс потока, который занимается поиском и подключением к менеджеру сервера. После получение индекса потока от ClientImpl, ClientConnectorThread вызывает getURLOfServer() у ClientImpl, в параметры которого передаёт индекс потока и ожидает ответа. При получении ответа по принятому адресу ServerImpl выполняется соединение помощью LipeRMI, и выполняется отправка задачи серверу, который уже готов к её выполнению.

Спустя некоторое время программе пользователя понадобились результаты вычислений задачи, отправленной ComputeEngine. Для получения результатов в ComputeEngine есть два метода – один синхронный (getResult()), другой асинхронный (getAsynResult()). В описанном примере используется синхронный вызов, который приостанавливает работу программы пользователя на время выполнения задачи и получения её результата. При вызове одного из методов получения результата ComputeEngine запрашивает ссылку на удаленный объект сервера, методом getServer() у потока ClientConnectorThread. После чего выполняется запрос серверу getResult() (если в ComputeEngine вызван метод getAsynResult(), то к серверу выполняется соединение по аналогичному запросу), возвращающий результат выполнения задачи.

Таким образом, пользователь получает возможность программировать программы, через которые с помощью ComputeEngine выполняются отправки заданий на выполнение. Важной особенности описанной реализации является то, что ComputeEngine позволяет программе работать далее, не загружая её на этапе поиска и соединения с сервером, выполнения задачи. Стоит отметить возможность пользователя самому управлять методом получения задачи, как синхронно, так и асинхронно.

На основе выше описанного примера было получено представление как работает экземпляр класса ComputeEngine и его компоненты, далее рассмотрим работу не менее важного компонента – это ClientImpl (Рисунок

6).

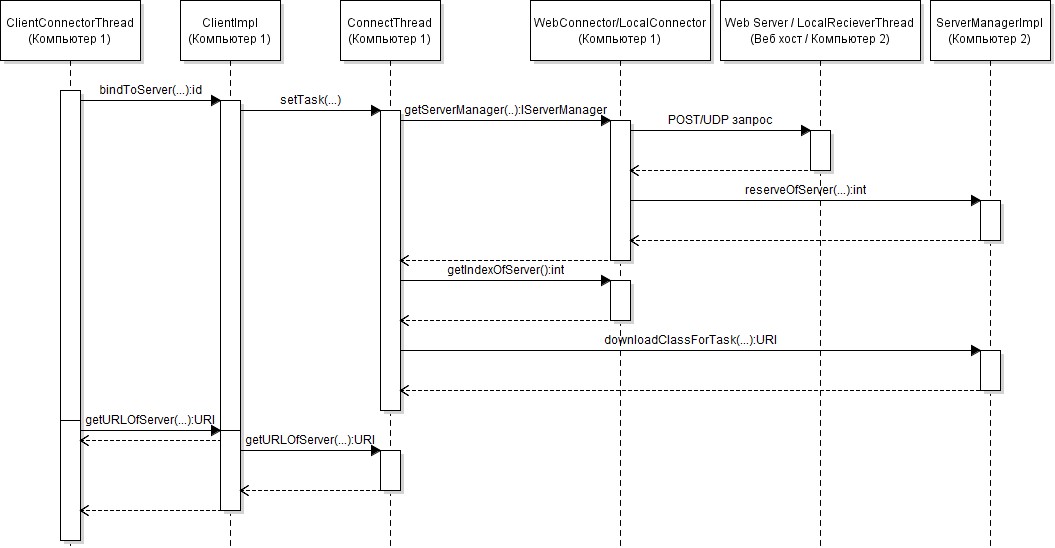


Рисунок 6 – Взаимодействие ClientImpl с ServerManagerImpl

В данном случае нет необходимости рассматривать действия со стороны пользовательской программы и ComputeEngine, возьмем только ClientConnectorThread и предположим, что от этого элемента был инициирован вызов у ClientImpl метода bindToServer, который был описан ранее. После вызова данного метода ClientImpl осуществляет поиск среди потоков ConnectThread свободный, если свободного потока не найдется, то запрос будет добавлен в очередь queueOfTasks типа Queue<String>, где запрос представляется в виде строки структуры:

“<путь к реализации задачи> <требуемая память>” (1)

Итак, когда ClientImpl находит свободный поток или пришла очередь задачи из очереди, методом свободного потока ConnectThread setTask(String classPath, int memoryForTask, ArrayList<URL> webServers), где classPath – путь к реализации задачи, memoryForTask – память для задачи и webServers – список адресов веб серверов, устанавливаются параметры и поток запускается.

При запуске, поток ConnectThread, изначально выполняет метод getServerManager(int countOfMemory) класса WebConnector, который формирует запрос в формате JSON с помощью класса WebRequest (не указан на рисунке, так как не выполняет важных функций), затем устанавливает связь с одним из веб серверов и отправляет POST запрос с JSON файлом. В ответ от сервера приходит файл формата JSON, содержащий список параметров для подключения к удаленному менеджеру серверов.

Форматы JSON-запросов и методы класса WebRequest создающие их.

Метод применяется для регистрации менеджера серверов в веб сервере.

public static String connect(int port, int memory)

{ return

String.format("{\"method\":\"connect\",\"port\":%d,\"memory\":%d}", port, memory);

}

Метод применяется для получения списка параметров менеджеров серверов в веб сервере.

public static String getURLOfServers(int memory)

{ return String.format("{\"method\":\"get\_servers\",\"memory\":%d}", memory);

}

Метод применяется для удаления параметров менеджера серверов с баз данных веб сервера.

public static String disconnect()

{ return "{\"method\":\"disconnect\"}"; }

Параметры менеджера серверов хранятся на веб сервере в базе данных, доступ к которой осуществляется через PHP скрипты, формирующий ответ на запрос. В нашем случае ответ будет содержать массив из трех параметров – это ip адрес компьютера 2, порт для подключения к ServerManagerImpl и максимальная память, предоставляемая им. Выборка нужных менеджеров осуществляется с помощью SQL запросов к базе данных со стороны PHP скрипта. Итак, WebConnector по полученным параметрам получает доступ к удаленному объекту с помощью LipeRMI и возвращает полученную ссылку ConnectThread.

Рассмотрим случай, когда у пользователя отсутствует соединение с интернетом или же нет доступных веб серверов. В данной ситуации WebConnector возвращает значение null, после чего ConnectThread повторяет такой же вызов метода getServerManager, но уже у LocalConnector, который работает с устройствами по локальной сети. LocalConnector выполняет широковещательную рассылку UDP-пакетов, содержащих одно число – количество требуемой памяти. В качестве приёмника UDP-пакетов выступает LocalRecieverThread, который принимает пакет, сравнивает полученное значение со значением, установленным ServerManagerImpl, и отправляет ответ с подтверждением в виде UDP пакета, содержащего номер порта менеджера серверов. LocalRecieverThread принимает подтвержденный ответ, получает ссылку на удаленный объект менеджера и отправляет её в качестве ответа ConnectThread.

Описанные классы WebConnector и LocalConnector имеют общий интерфейс Connector, а также при вызове метода getServerManager резервируют в нём сервер методом reserveOfServer(String host,int countOfMemory), где host – это адрес текущего компьютера, от которого идет запрос.

ConnectThread, получив ссылку на удаленный ServerManagerImpl, получает индекс выделенного сервера от WebConnector/LocalConnector, затем упаковывает в Zip-архив файлы, на которые указывает переменная classPath,

и выполняет вызов метода удаленного менеджера серверов downloadClassForTask(byte[] zipOfClassFiles, int indexOfServer), в котором передает массив байтов созданного архиватором ZipArchivator и индекс сервера. В ответ ConnectThread получает URI для подключения к выделенному серверу.

Через некоторое время ClientConnectorThread выполняет запрос getURLOfServer(int index) у ClientIml, тот по индексу index находит поток и делает запрос getURLOfServer, полученный URI клиент отправляет ClietnConnectorThread. После этого этапа клиент не участвует в дальнейших действиях между приложением пользователя и сервера.

Также следует отметить, что данная реализация выполняет выделение сервера в любом случае. Так если нет интернета, то поиск происходит по локальной сети, однако, если в локальной сети не нашлось задачи, она будет запущена на сервере компьютера данного пользователя. Таким образом обеспечивается гарантия выполнимости поставленной задачи.

Итак, на основе рассмотренных примеров и диаграмм был представлен основной функционал программы CRDS, а именно, класс ComputeEngine, с помощью которого пользователь может отправить свою задачу на выполнение удаленной машине, без необходимости ожидания её завершения. Были описаны основные методы классов ComputeEngine, ClientImpl, ClientConnectorThread, ConnectThread, LocalConnector, WebConnector, WebRequest, ServerManagerImpl, ServerImpl.

При описании алгоритмов взаимодействия между удаленными объектами, не было учтено одно, после завершения работы ServerImpl и отправки пользователю результата, сервер остается запущенным, в силу того, что при привязке ссылки на объект в RMI регистре, этот объект находится в отдельном потоке, который не является “демоном” и, следовательно, программа не завершится, пока её не завершат. Для закрытия программы ServerImpl применяется класс ScheduledTask, который отправляет запрос getPortsOfServers() локальному ServerManagerImpl, от которого получает список портов от запущенных серверов. Затем, последовательно перебирая порты и подключаясь к серверам с помощью LipeRMI, ScheduledTask проверяет статусы серверов (класс ServerStatus) и если статус равен COMPLETE или ERROR, то индекс данного сервера помещается в список. После описанной процедуры список отправляется ServerManagerImpl методом closeServers(ArrayList<Integer> listOfServers), который по указанным индексам закрывает приложения серверов, вызовом метода класса ProcessController destroy(), после чего сбрасываются все настройки класса ServerInfo, используемого менеджером серверов как информационного объекта и содержащего в списке listOfServers. Описанный сценарий изображен на рисунке 7.

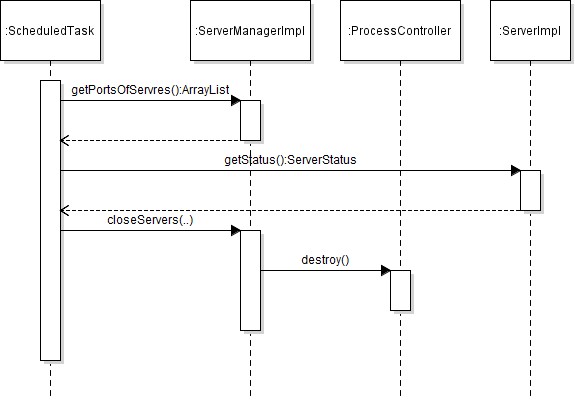


Рисунок 7 – Процесс проверки и завершения работы ServerImpl

Следует учесть, то что ScheduledTask запускается таймером из класса CRDS с определенным константой интервалом. Также все файлы, которые передаются со стороны клиента и используются сервером, по завершению работы автоматически удаляются.

Помимо основного функционала в приложении CRDS имеется

возможность контроля. Для того чтобы получать сведения о системе или изменять её параметры был разработан класс CRDSController.

Экземпляр объекта CRDSController позволяет программисту вызывать следующие методы для взаимодействия с системой CRDS:

а) boolean setMemoryForServers(int countOfMemory) – установка памяти, выделяемой серверами по умолчанию. Возвращает false в случае ошибки или недопустимого значения параметра.

б) boolean setMemoryForServer(int countOfMemory, int indexOfServer) – установка количества памяти для сервера под индексом indexOfServer.

Возвращает false в случае ошибки или недопустимого значения параметра.

в) int getMemoryForServers() - получение количества памяти

установленной для серверов по умолчанию.

г) int getMemoryForServer(int indexOfServer) – получение количества

памяти для сервера под указанным индексом.

д) boolean setMemoryForTasks(int countOfMemory) – установка количества памяти по умолчанию для задач пользователя. Возвращает false в случае ошибки или недопустимого значения параметра.

е) int getMemoryForTasks() – получение количества памяти по

умолчанию для задач пользователя.

ж) boolean setCountOfServers(int countOfServers) – установка нового значения количества серверов, выделяемых менеджером серверов.

Возвращает false в случае ошибки или недопустимого значения параметра.

и) int getCountOfServers() – получение количества серверов,

выделяемых менеджером серверов.

к) boolean setCountOfTasks(int countOfTasks) – установка нового значения количества задач, обрабатываемых клиентом без очереди.

Возвращает false в случае ошибки или недопустимого значения параметра.

л) int getCountOfTasks() – получение количества задач, обрабатываемых

клиентом без очереди.

м) boolean setListOfWebServers(ArrayList<URL> webServers) – установка нового списка веб серверов. Возвращает false в случае ошибки или недопустимого значения параметра.

н) ArrayList<URL> getListOfWebServers() – получение списка веб

серверов, используемых системой.

п) boolean addWebServer(URL webServer) – добавление нового веб сервера в существующий список системы. Возвращает false в случае ошибки или недопустимого значения параметра.

р) boolean removeWebServer(URL webServer) – удаление указанного веб сервера из списка системы. Возвращает false в случае ошибки или недопустимого значения параметра.

с) ServerStatus getStatusOfServer(int indexOfServer) – получение статуса

сервера по указанному индексу.

т) ArrayList<ServerStatus> getStatusesOfServers() – получение списка статусов всех серверов менеджера серверов.

у) TaskStatus getStatusOfTask(int indexOfTask) – получение статуса

задачи по указанному индексу.

ф) ArrayList<TaskStatus> getStatusesOfTasks() – получение списка

статусов всех задач обрабатываемых клиентом.

void setLocalSever(boolean flag) – установка флага применения в системе поиска по локальной сети.

х) boolean isUsedLocalServers() – получение значения флага

определяющего используется ли в системе поиск по локальной сети.

Все описанные выше методы позволяют изменять состояние компонентов системы. Так как данный инструмент является дополнением к системе и не является основным инструментом, принципы работы которого описаны в разделе 2.2, то полный обзор логической структуры отсутствует в данном контексте. Реализации классов ClientImpl и ServerManagerImpl приведены в ПРИЛОЖЕНИЕ C.

2.4.5 Тестирование

Тестирование системы CRDS проводилось на реальных устройствах, которых было 4. Первый компьютер выступал в качестве раздающего задачи, т.е. в роли клиента. Остальные устройства использовались для выделения под задачи серверов. Для удобства обозначим компьютеры, принимавшие участие в тестировании, узлами. Так, узел 1 – это клиент, который связан с узлами 2 и 3 через локальную сеть с помощью экранированной витой пары 3 категории, пропускная способность которой составляет 100Мбит/c с частотой 16МГц.

Узлы 1, 2 и 3 соединены локальной сетью с помощью маршрутизатора. Узел 4 соединён с узлом 1 через интернет, пропускная способность которого, со стороны первого узла – 50Мбит/с, а со стороны второго 36Мбит/c.

Технические характеристики узлов: а) Узел 1.

1. Процессор: Intel Core i3-2328M 2.2GHz, 2 ядра, 64 разрядный.
2. Память: 4Gb.
3. Операционная система: Windows 10 x64.
4. Адрес в локальной сети: 192.168.1.1.

б) Узел 2.

1. Процессор: Intel Core Duo E7200 2.53GHz, 2 ядра, 32 разрядный.
2. Память: 2Gb.
3. Операционная система: Windows 7 x84.
4. Адрес в локальной сети: 192.168.1.2.

в) Узел 3.

1. Процессор: Intel Core i5 3230M 2.6GHz, 2 ядра, 64 разрядный.
2. Память: 4Gb.
3. Операционная система: Windows 10 x64.
4. Адрес в локальной сети: 192.168.1.3.

г) Узел 4.

1. Процессор: Intel Pentium D 2.2GHz, 2 ядра, 64 разрядный.
2. Память: 4Gb.
3. Операционная система: Windows 7 x64.
4. Адрес в глобальной сети: 212.15.211.3.

Для тестирования была написана программа на Java в виде класса Test, который запускался через командную строку с параметром, определяющим количество задач выделяемые для тестирования.

В качестве задачи выступает, реализующий интерфейс Task, класс Permutation, который выполняет следующие задачи:

а) Принимает на вход конструктора строку symbols, служащую алфавитом, и число len – размер генерируемой строки str.

б) Первоначально строка str состоит из len первых символов алфавита

symbols.

в) Производится перебор всевозможных строк. Для этого берется каждый символ строки и изменяется на символ, который стоит в алфавите после него, тем самым полученная строка приравнивается к str. Процесс продолжается, пока рассматриваемый символ не будет являться последним в алфавите symbols, после чего осуществляется такой же перебор следующего символа строки str. Функция выполняется пока не будет получена строка, у которой все символы равны последнему символу алфавита symbols. В конце работы алгоритма выводится время, затраченное на его выполнение в миллисекундах.

Программа Test запускает на выполнение задачу Permutation указанное количество раз, при этом для каждой задачи создается экземпляр класса ComputeEngine, который передает её системе CRDS. Система CRDS принимает задачи и распределяет между узлами 2-4.

Для тестирования программа Test была запущена со следующими параметрами:

* java –jar test.jar 1 – требуется распределить 1 задачу;
* java –jar test.jar 3 – требуется распределить 3 задачи;
* java –jar test.jar 6 – требуется распределить 6 задач;
* java –jar test.jar 12 – требуется распределить 12 задач.

Перед каждым запуском у систем CRDS на узлах 2-4 были изменены параметры, таким образом, чтобы во время второго теста – каждый узел получил одну задачу, третий тест – 2 задачи на узел и 4 задачи на узел во время четвертого теста.

Первый тест выполнен с целью продемонстрировать работу системы через сеть интернет. Таким образом узел 1 передает через интернет задачу узлу 4 и принимает результат. Запуск и завершение программы Test с параметром 1 на первом узле продемонстрирован на рисунке 8, данные выводимые системой CRDS на узле 4 изображены на рисунке 9.

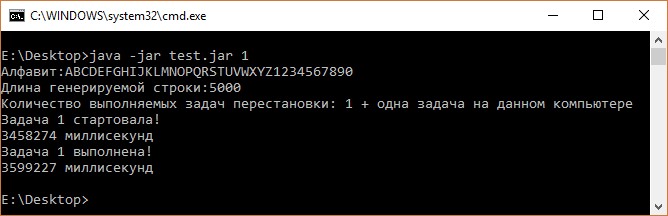


Рисунок 8 – Результат работы программы Test с параметром 1

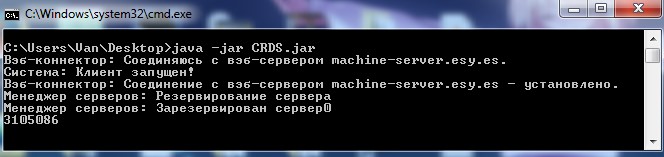


Рисунок 9 – Работа программы CRDS на узле 4 во время запуска программы Test с параметром 1

Также перед запуском первого теста система CRDS была запущена на узле 1 через командную строку (Рисунок 10), что сделал и пользователь узла 1 (Рисунок 9). Таким образом имеется возможность наблюдать за сообщениями системы CRDS.

Как мы видим из рисунков 8-10 тест прошел успешно. При этом система узла 1 соединилась с менеджером серверов узла 4 по адресу 212.15.211.3:58599. После чего задача Permutation по завершению своей работы вывела сообщение 3105086 – количество миллисекунд, затраченных на выполнение задачи. Это сообщение было перехвачено и выведено ProcessController-ом системы узла 4.

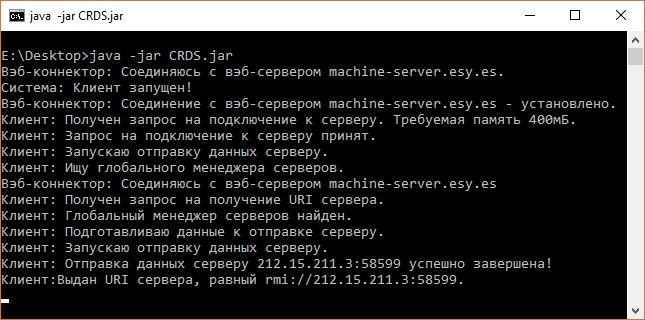


Рисунок 10 – Работа системы CRDS во время запуска программы Test с

параметром 1

На стороне клиента программа Task вывела сначала время задачи, выполненной на узле 1 – это 3458274 миллисекунд, после чего ожидала завершения работы сервера. По завершению работы общее время работы программы составило 3599227 миллисекунд. Вывод – благодаря системе CRDS и инструменту ComputeEngine пользователь узла 1 решил примерно за

час две задачи, суммарное время выполнения которых 6 704 313 миллисекунд или 111 минут.

Скриншоты работы программы CRDS узлов 1-4 во время тестов 2-4 не указаны в данном контексте, в следствие большого объема информации выводимой системой. Однако, результаты работы каждого сервера были зафиксированы.

После проведения тестов 2-4, результаты тестов показаны на рисунках 11-16. Состоящих из скриншота работы программы Test и диаграммы

времени выполнения задач на серверах каждого узла.

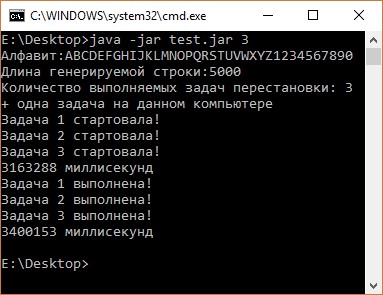


Рисунок 11 – Скриншот работы программы Test с параметром 3



0

1000000

2000000

3000000

4000000

узел 1

узел 2 сервер 1

узел 3 сервер 1

узел 4 сервер 1

узел 1 всего

Тест 1

Рисунок 12 – Диаграмма времени выполнения задач на разных узлах в миллисекундах после запуска программы Test с параметром 3

После выполнения второго теста система показала результаты, указанные на диаграмме рисунка 12, узел 1 – это данные по работе алгоритма Permutation на компьютере программы пользователя, а узел 1 всего – общее время работы программы Test. В таком случае, на системах CRDS узлов 2-4 был выделен только один сервер, каждый из которых был зарезервирован клиентом узла 1 и получил задачу. В результате программа Task выполнила 4 задачи Permutation за время, сопоставимое выполнению одной задачи.

В данном тесте, учитывая особенности выполнения программ в JVM, сервера узлов 2-4 использовали около 400 Мбайт памяти и 25-35% процессорного времени. Такие настройки системы (выделение одного сервера) не мешают пользователю во время работы с другими приложениями.

Для того, чтобы проверить, как система будет расходовать ресурсы компьютер, была увеличена нагрузка увеличением количества серверов узлов в тестах 3 и 4, результаты которых представлены в виде диаграмм, которые изображены на рисунках 14 и 16.

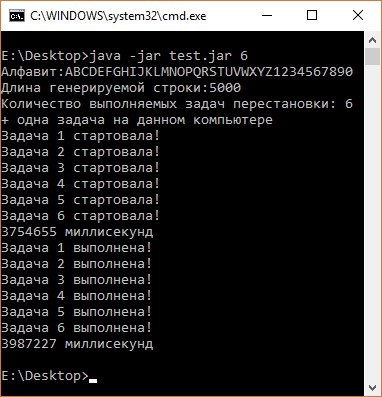


Рисунок 13 - Скриншот работы программы Test с параметром 6



0

1000000

2000000

3000000

4000000

5000000

узел 1

узел 2 сервер 1

узел 2 сервер 2

узел 3 сервер 1

узел 3 сервер 2

узел 4 сервер 1

узел 4 сервер 2

узел 1 всего

Тест 2

Рисунок 14 – Диаграмма времени выполнения задач на разных узлах в миллисекундах после запуска программы Test с параметром 6

Из проведенного третьего теста следует вывод, что при увеличении числа принимаемых задач программой CRDS, вычисления серверами будут проводится медленно, так среда Java автоматические управляет распределением процессорного времени среди запущенных ей программ. Таким образом процессор для каждого сервера был загружен на 15-25% и расходовал около 400 Мбайт памяти.

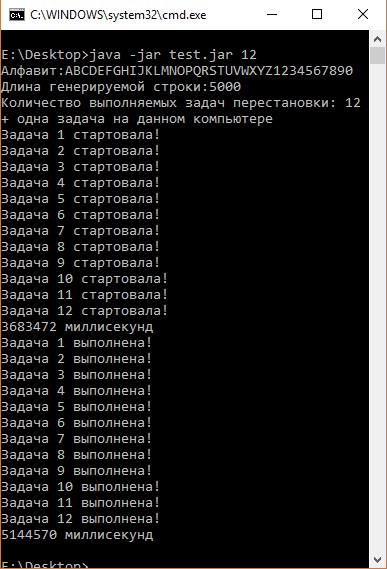


Рисунок 15 - Скриншот работы программы Test с параметром 6

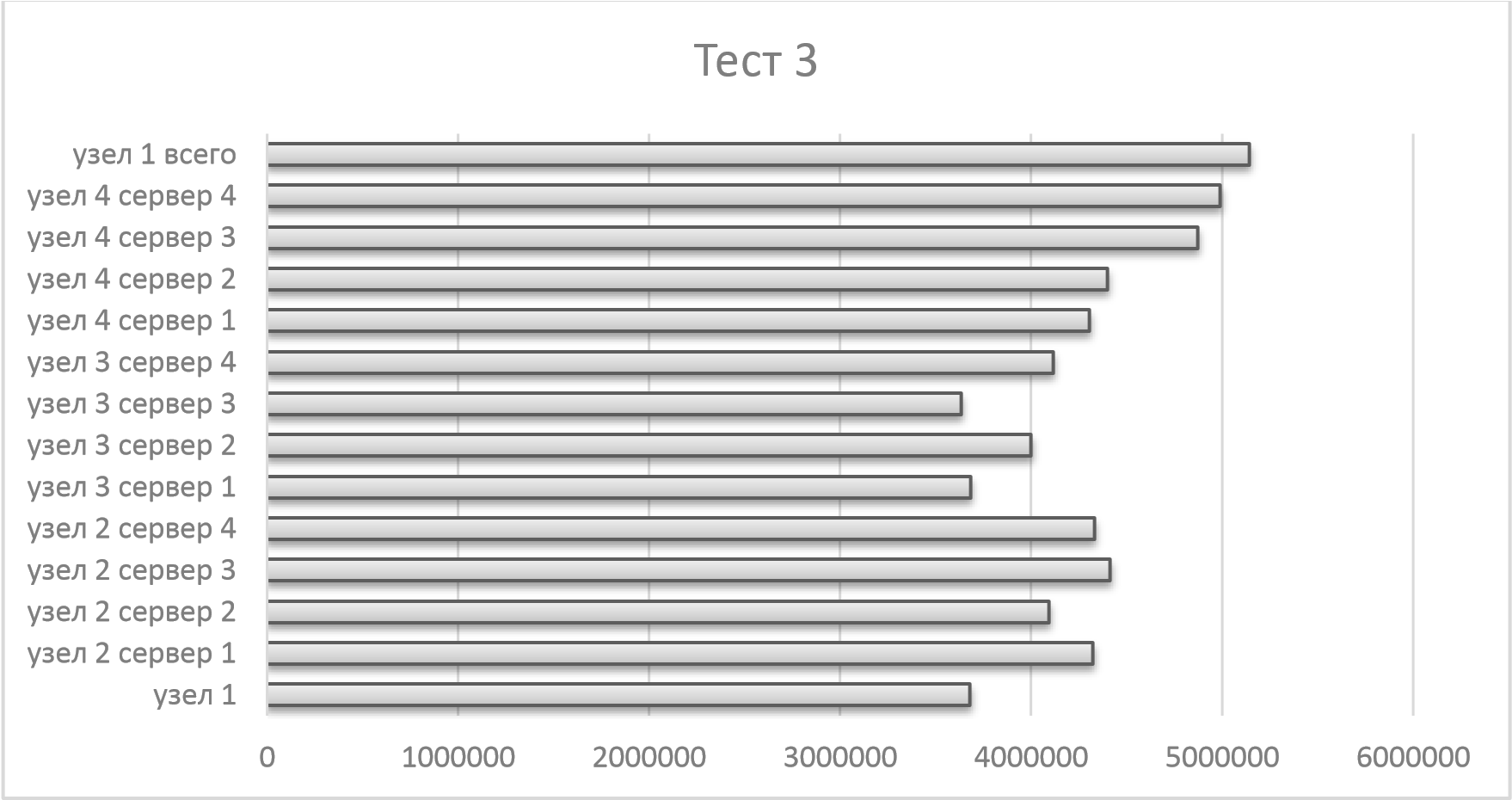


Рисунок 16 – Диаграмма времени выполнения задач на разных узлах в миллисекундах после запуска программы Test с параметром 12

Вывод полученный после проведения третьего теста был подтвержден результатами четвертого теста. Несмотря на то, что задачи в четвертом тесте выполнялись гораздо медленнее (загрузка процессора 6-12%), результат полученный пользователем узла 1 был вычислен быстрее, чем если он был получен вычислением 13 задач на одном компьютере. Данный результат демонстрирует преимущества распределенных систем, как систем предоставляющих вычислительную мощность. И даже в случае использования четырех серверов, пользователь не почувствует дискомфорта при работе за компьютером.

На этом этап тестирования программы завершен, после которого можно сделать вывод, что разработанная программа выполняет поставленные перед ней задачи, а распределенная система обеспечивает бесперебойную работу и наименьшую нагрузку (за исключением использования оперативной памяти).

Исходные коды программ Test и Permutation используемых в тесте, приведен в ПРИЛОЖЕНИЕ B.

2.5 Безопасность

Распределенные системы в силу того, что организуют сетевые соединения между компьютерами, должны гарантировать пользователю защиту его компьютера от атак со стороны злоумышленников, а также сохранность данных при передаче между устройствами.

В разработанной системе опасность может предоставлять, не только перехват информации клиента или сервера третьим лицом, но и возможность использования программы для загрузки на удаленный компьютер зловредной задачи. Под зловредной задачей будем понимать программный код, способный выполнить, прочитать, изменить, удалить или загрузить из интернета файл, а также переполняющий оперативную память компьютера.

Для решения данных проблем в Java присутствуют специальные средства защиты. В случае, когда пользователь попытается передать удаленному серверу программу, которая загружает и выполняет файл или же пытается получить доступ к файлам пользователя с целью удаления или изменения, то на такой случай в Java присутствует менеджер безопасности

(SecurityManager), который можно получить методом System.getSecurityManager() или установить System.setSecurityManager(new

SecurityManager()). С помощью менеджера безопасности можно задавать параметры безопасности внутри программного кода или же использовать специальные файлы policy, которые содержат инструкции с ограничениями для указанного каталога.

В CRDS применяется SecurityManager, в первую очередь следует отметить что этого требует технология RMI при построении приложения. Для каждого компонента системы Client, CRDS и Server, для которых сгенерированы файлы policy с именами: crds\_policy, server\_policy, client\_policy. Где важным является client\_policy, который содержит такой параметр, как java.io.FilePermission "${user.dir}", "read" означающий, что серверу доступна для чтения только пользовательская дерриктория, из которой запущена система CRDS. Остальные файлы будут недоступны.

Второй случай поведения зловредной задачи – это перегрузка компьютера путем переполнения оперативной памяти решается архитектурой системы CRDS, а именно указанный максимум оперативной памяти (в мегабайтах), которая будет доступна одному серверу. Однако тут стоит помнить, что пользователь может поменять это значение. Которого зловредной задаче окажется достаточно для перегрузки компьютера. Также попытка загрузить процессор не выйдет по причине регулировки со стороны JVM. Защита во время сетевых соединениях между удаленными объектами обеспечивается технологией RMI.

Система CRDS гарантирует пользователю безопасность, ограничивая доступ как к файлам операционной системы, так и объем используемых вычислительных ресурсов.

ЗАКЛЮЧЕНИЕ

В ходе дипломной работы была разработана программа CRDS, а также инструменты взаимодействия пользователя с этой программой – ComputeEngine и CRDSController. Программа CRDS представляет собой распределенную систему поддерживающую связь с такими же программами на удаленных компьютерах и позволяющую использовать, а также предоставлять пользователям, вычислительные ресурсы компьютера.

Программа CRDS минимально загружает процессор (от 0 до 30%) и потребляет минимум оперативной памяти в пассивном режиме (в активном режиме потребление зависит от настроек системы).

ComputeEngine – является инструментом для взаимодействия пользователя с программой CRDS и получением её функционала, при этом данный класс позволяет на высоком уровне абстракции поставить задачу удаленному компьютеру и получить результат.

CRDSController – инструмент для управления и контроля системы CRDS, также может быть использован для разработки отдельного программного обеспечения с применением графического интерфейса.

Программа прошла успешное тестирование и показала хорошие результаты, при больших нагрузках.

Сфера применения программы, написанной с помощью ComputeEngine зависит только от программиста, в основном это области, требующие больших вычислительных мощностей – научные сферы, моделирование, торговые биржи и другие.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Радченко Г. И. Распределенные вычислительные системы// - Челябинск:Фотохудожник, 2012. – C. 184.
2. Косяков М. С. Введение в распределенные вычисления// -СПб: НИУ ИТМО, 2014. – С. 155.
3. BOINC документация [Электронный ресурс]. URL: http:// boinc.berkeley.edu/trac/wiki/ProjectMain (дата обращения 10.05.2017).
4. Применение LipeRMI [Электронный ресурс]. URL: http:// lipermi.sourceforge.net/documentation.php (дата обращения 25.04.2017) .
5. Таненбаум Э., Уэзеролл Д. Компьютерные сети. 5-е изд.// — СПб.: Питер, 2012. — С. 960.
6. Оберг Роберт Дж. Технология СОМ+. Основы и

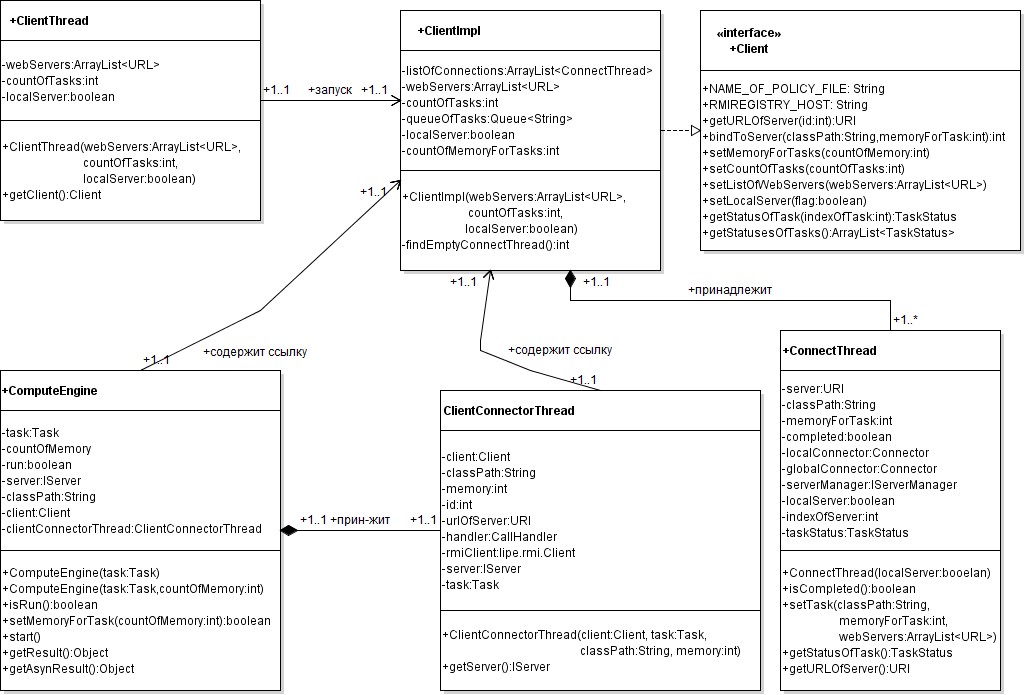
программирование.//

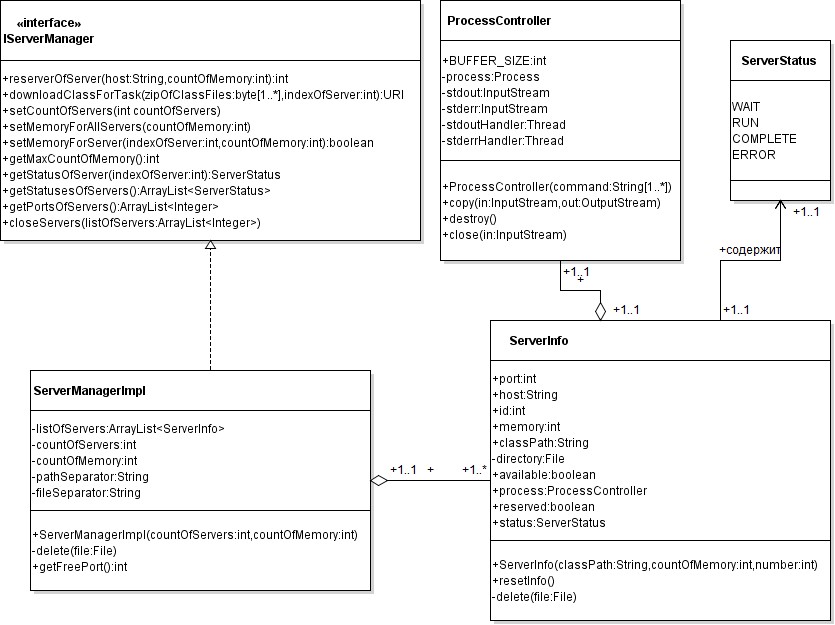
- М.: Издательский дом "Вильямс", 2000. – С. 480.

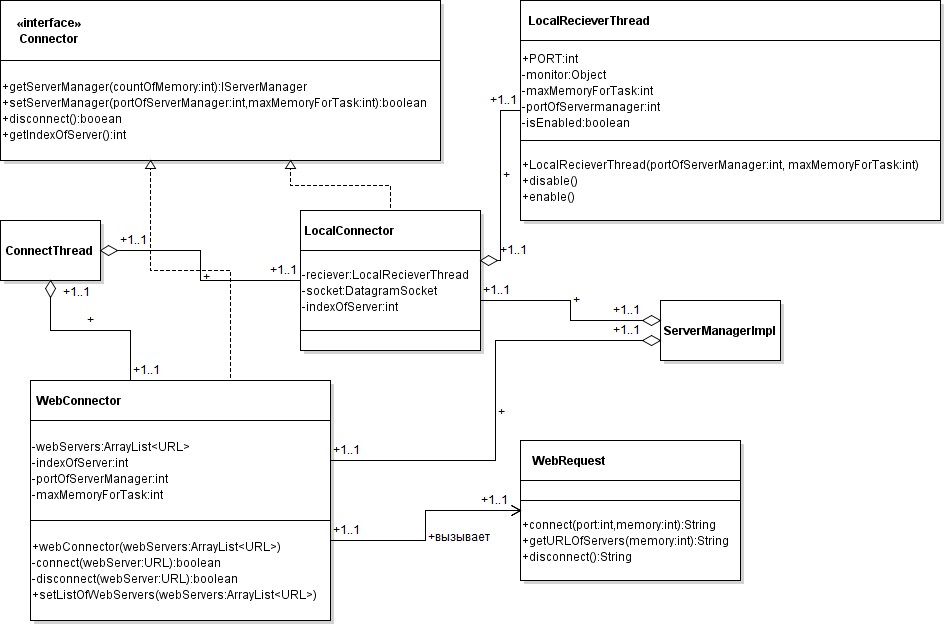
1. Обзор работы с RMI [Электронный ресурс] URL: https:// docs.oracle.com/ javase tutorial/rmi/overview.html (дата обращения 4.03.2017).
2. Таненбаум Э. М. ван Стеен. Распределенные системы. Принципы и парадигмы //— СПб.: Питер, 2003. — С. 877.

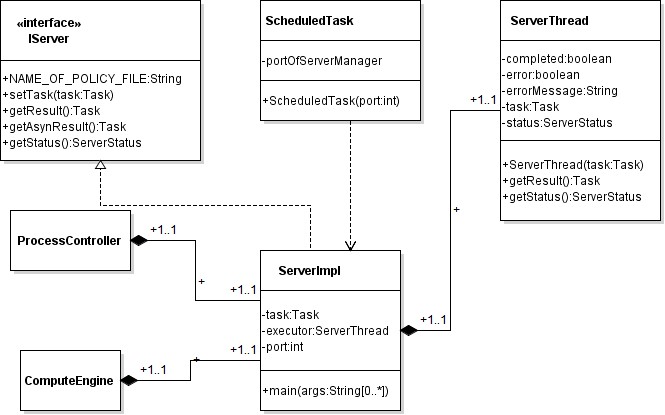
ПРИЛОЖЕНИЕ А

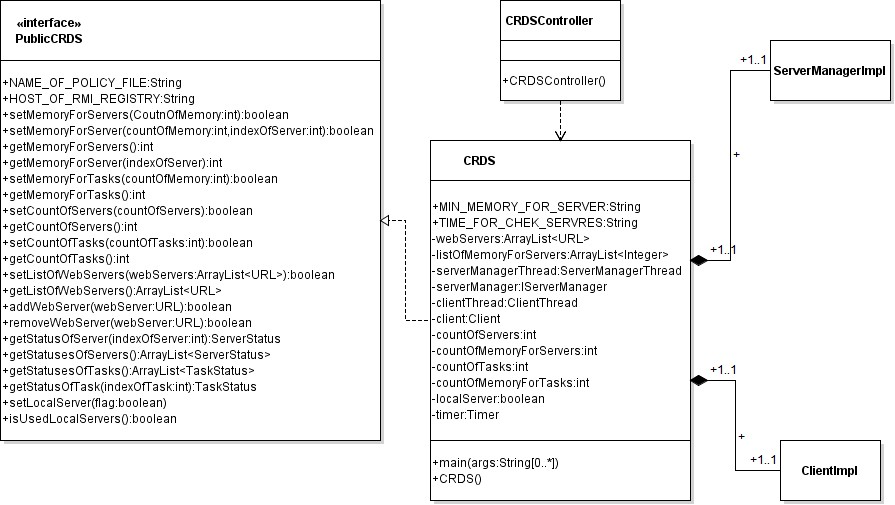
UML диаграммы классов системы CRDS











ПРИЛОЖЕНИЕ B

Исходный код классов Test и Permutation

package test;

import compute\_engine.ComputeEngine; import crds.CRDSController; import java.rmi.RemoteException; import java.util.ArrayList; public class Test {

public static void main(String[] args) throws RemoteException { CRDSController system=new CRDSController(); int countOfTasks=Integer.parseInt(args[0]);

//установка в системе CRDS нового значения - количество задач system.setCountOfTasks(countOfTasks);

String ABC="ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"; int len=5000;

long timer=System.currentTimeMillis();

ArrayList<ComputeEngine> listOfComputers=new ArrayList<>();

System.out.println("Алфавит:"+ABC);

System.out.println("Длина генерируемой строки:"+len);

System.out.println("Количество выполняемых задач перестановки: "

+countOfTasks+" + одна задача на данном компьютере");

for(int i=1;i<=countOfTasks;i++)

{

Permutation perm=new Permutation(ABC,len);

//создание и запуск вычислителя с задачей perm

//для которой необходимо 400Мбайт оперативной памяти ComputeEngine cmp=new ComputeEngine(perm,400); cmp.start();

System.out.println("Задача "+i+" стартовала!"); listOfComputers.add(cmp);

}

Permutation pm=new Permutation(ABC, len);

System.out.println(System.currentTimeMillis()-timer+" миллисекунд"); int index=1;

for(ComputeEngine computer:listOfComputers)

{

//синхронное получение решенной задачи

Permutation perm=(Permutation)computer.getResult(); System.out.println("Задача "+index+" выполнена!"); index++; }

System.out.println(System.currentTimeMillis()-timer+" миллисекунд");

}

}

package test;

import task.Task;

public class Permutation implements Task{

String symbols, str=""; int n;

public Permutation(String symbols, int len)

{

this.symbols=symbols; this.n=len; }

@Override public void execute() { long timer=System.currentTimeMillis(); char last\_sym=symbols.charAt(symbols.length()-1);

for(int i=0;i<n;i++) str+=symbols.charAt(0);

//Перебор строки с длинной n и алфавитом symbols for(int i=0;i<n;i++) while(str.charAt(i)!=last\_sym)

{

String new\_str=""; for(int j=0;j<n;j++) if(j!=i)

new\_str+=str.charAt(j);

else

new\_str+=next(str.charAt(i)); str=new\_str;

}

System.out.println(System.currentTimeMillis()-timer);

}

public char next(char x)

{

int index=symbols.indexOf(x); if(index!=symbols.length()-1)

return symbols.charAt(index+1);

else

return symbols.charAt(0);

} }

ПРИЛОЖЕНИЕ С

Исходный код классов ClientImpl и ServerManagerImpl

package client; import java.net.URI; import java.net.URL; import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject;

import java.util.ArrayList; import java.util.PriorityQueue; import java.util.Queue; import task.TaskStatus;

/\*\* Класс служит для создания соединения с удаленным сервером сервером

\*

\* @author Vitaly Kuzmin

\*/

class ClientImpl extends UnicastRemoteObject implements Client{ /\*\* Свойство - список выполняемых подключений к серверу\*/ private ArrayList<ConnectThread> listOfConnections;

/\*\* Свойство - список URL адресов веб серверов\*/ private ArrayList<URL> webServers;

/\*\* Свойство - количество задач, которые обрабатываются одновременно\*/ private int countOfTasks; /\*\* Свойство - очередь задач\*/ private Queue<String> queueOfTasks;

/\*\* Свойство - использовать ли сервера локальной сети\*/ private boolean localServer;

/\*\* Свойство - количество памяти для задач по умолчанию\*/ private int countOfMemoryForTasks;

/\*\* Создает новый объект с заданными значениями

* @param webServers - список URL адресов веб серверов
* @param countOfTasks - количество задач, обрабатываемых одновременно
* @param localServer - использовать ли сервера локальной сети
* @throws RemoteException

\*/

public ClientImpl(ArrayList<URL> webServers, int countOfTasks,

boolean localServer) throws RemoteException{

this.localServer=localServer; listOfConnections=new ArrayList<>(); queueOfTasks=new PriorityQueue<String>(); this.webServers=webServers;

for(int i=0;i<countOfTasks;i++)

{

ConnectThread x=new ConnectThread(localServer);

listOfConnections.add(x);

}

}

@Override public synchronized URI getURLOfServer(int index) throws RemoteException {

System.out.println("Клиент: Получен запрос на получение URI сервера.");

ConnectThread connection=listOfConnections.get(index);

if(connection.isCompleted())

{

URI url=connection.getURLOfServer(); System.out.println(url.getHost()+url.getPort()); if(listOfConnections.size()>countOfTasks && index==(listOfConnections.size()-1))

{

listOfConnections.remove(index);

System.out.println("Клиент:Выдан URI сервера, равный rmi://"

+url.getHost()+":"+url.getPort()+"."); return url;

}

connection=listOfConnections.set(index, new ConnectThread(localServer)); if(!queueOfTasks.isEmpty())

{

String classPath=queueOfTasks.poll(); int memory=Integer.parseInt(

classPath.substring(classPath.lastIndexOf(" "))); classPath=classPath.substring(0, classPath.lastIndexOf(" "));

connection.setTask(classPath, memory, webServers);

}

System.out.println("Клиент:Выдан URI сервера, равный rmi://"

+url.getHost()+":"+url.getPort()+"."); return url;

}

System.out.println("Клиент:URI сервера не выдан."); return null;

}

@Override

public synchronized int bindToServer(String classPath, int memoryForTask) throws RemoteException { int index=findEmptyConnectThread();

System.out.println("Клиент: Получен запрос на подключение к серверу. "

+ "Требуемая память "+memoryForTask+"мБ."); if(index==-1)

{

queueOfTasks.add(classPath+" "+memoryForTask);

System.out.println("Клиент: Запрос на подключение к серверу "

+ "добавлен в очередь.");

}else

{

listOfConnections.get(index).setTask(classPath, memoryForTask, webServers);

System.out.println("Клиент: Запрос на подключение к серверу "

+ "принят.");

}

return index; }

/\*\* Функция для поиска потока, кторый не выполняется в данный момент

\* @return Возвращает индекс потока

\*/

private synchronized int findEmptyConnectThread()

{

for(int i=0;i<listOfConnections.size();i++) if(!listOfConnections.get(i).isAlive()

&& !listOfConnections.get(i).isCompleted()) return i; return -1;

}

@Override

public synchronized void setMemoryForTasks(int countOfMemory) throws RemoteException {

countOfMemoryForTasks=countOfMemory;

System.out.println("Клиент: Установлено новое значение памяти(по умолчанию) равное "

+countOfMemory+"мБ для всех задач.");

}

@Override

public synchronized void setCountOfTasks(int countOfTasks) throws RemoteException { ConnectThread connection;

System.out.println("Клиент: Принят запрос на установку количества обрабатываемых задач равное "

+countOfTasks+".");

if(countOfTasks>this.countOfTasks)

{

for(int i=listOfConnections.size();i<countOfTasks;i++)

{

connection=new ConnectThread(localServer);

if(!queueOfTasks.isEmpty())

{

String classPath=queueOfTasks.poll(); int memory=Integer.parseInt( classPath.substring(classPath.lastIndexOf(" "))); classPath=classPath.substring(0, classPath.lastIndexOf(" "));

connection.setTask(classPath, memory, webServers);

}

listOfConnections.add(connection);

}

}else if(countOfTasks<this.countOfTasks)

{

for(int i=listOfConnections.size()-1;i>=0;i--)

{

connection=listOfConnections.get(i);

if(!connection.isAlive() && !connection.isCompleted()) listOfConnections.remove(i);

else break;

}

}

this.countOfTasks=countOfTasks;

System.out.println("Клиент: Установлено новое значение количества обрабатываемых задач равное "

+countOfTasks+".");

}

@Override

public synchronized void setListOfWebServers(ArrayList<URL> webServers) throws RemoteException {

System.out.println("Клиент: Установлен новый список вэб-серверов."); this.webServers=webServers;

}

@Override

public synchronized void setLocalServer(boolean flag) throws RemoteException {

if(flag)

System.out.println("Клиент: Использование локальных серверов разрешено."); else

System.out.println("Клиент: Использование локальных серверов запрещено."); localServer=flag;

}

@Override

public synchronized TaskStatus getStatusOfTask(int indexOfTask) throws RemoteException { if(indexOfTask<listOfConnections.size())

return listOfConnections.get(indexOfTask).getStatusOfTask(); return null;

}

@Override

public synchronized ArrayList<TaskStatus> getStatusesOfTasks() throws RemoteException {

ArrayList<TaskStatus> listOfStatuses=new ArrayList<>(); for(ConnectThread connection:listOfConnections) listOfStatuses.add(connection.getStatusOfTask()); if(!queueOfTasks.isEmpty())

{

for(int i=0;i<queueOfTasks.size();i++)

listOfStatuses.add(TaskStatus.WAIT);

}

return listOfStatuses;

}

}

package server;

import connector.WebConnector; import java.io.BufferedOutputStream;

import java.io.File; import java.io.FileOutputStream; import java.io.IOException; import java.io.OutputStream; import java.net.MalformedURLException;

import java.net.ServerSocket; import java.net.URI;

import java.net.URISyntaxException; import java.net.URL; import java.util.ArrayList; import java.util.logging.Level; import java.util.logging.Logger; import zip.ZipArchivator;

/\*\* Класс служит для управления серверами, предназначенными для удаленных

* компьютеров
* @author Vitaly Kuzmin

\*/

public class ServerManagerImpl implements IServerManager{ /\*\*Свойство - список информационных объектов серверов\*/ private ArrayList<ServerInfo> listOfServers;

/\*\*Свойство - количество серверов\*/ private int countOfServers;

/\*\*Свойство - количество памяти\*/ private int countOfMemory;

/\*\*Свойство - разделитель путей в переменной среды classpath\*/ private String pathSeparator=System.getProperty("path.separator");

/\*\*Свойство - разделитель в пути к фалу/каталогу\*/ private String fileSeparator=System.getProperty("file.separator");

/\*\*Создаёт новый объект с заданными значениями

* @param countOfServers - количество серверов
* @param countOfMemory - количество памяти

\*/

public ServerManagerImpl(int countOfServers, int countOfMemory)

{

new File("server\_classes").mkdir(); this.listOfServers=new ArrayList<>(); this.countOfMemory=countOfMemory; this.countOfServers=countOfServers;

for(int i=0; i<countOfServers; i++)

{

File file=new File("server\_classes"+fileSeparator+"server"+i); file.mkdir();

ServerInfo server=new ServerInfo(file.getAbsolutePath(),countOfMemory,i); listOfServers.add(server);

}

}

@Override

public synchronized URI downloadClassForTask(byte[] zipOfClassFiles, int indexOfServer) {

ServerInfo server=listOfServers.get(indexOfServer); if(server.reserved && server.available)

{

String zipPath=server.classPath+"server\_pack.zip"; try {

File zipFile=new File(zipPath); if(zipFile.exists()) zipFile.delete(); zipFile.createNewFile();

OutputStream out = new BufferedOutputStream(new FileOutputStream(zipPath)); out.write(zipOfClassFiles);

out.close();

ZipArchivator.unZip(zipPath, server.classPath); File classDirectory=new File(server.classPath); for(File file:classDirectory.listFiles())

{

if(file.getName().toLowerCase().endsWith(".jar"))

server.classPath+=pathSeparator+server.classPath+fileSeparator+file.getName();

}

server.classPath+=pathSeparator+System.getProperty("java.class.path"); zipFile.delete();

} catch (IOException ex) {

Logger.getLogger(ServerManagerImpl.class.getName()).log(Level.SEVERE, null, ex);

}

}

server.port=getFreePort(); server.available=false;

String[] command=new String[]{"java",

"-Xmx"+server.memory+"m",

"-cp",

server.classPath,

ServerImpl.class.getName(),

String.valueOf(server.port)};

System.out.println("Отправил на порт "+server.port); server.status=ServerStatus.RUN;

server.process=new ProcessController(command); try {

return new URI("rmi://"+server.host+":"+server.port);

} catch (URISyntaxException ex) {

Logger.getLogger(ServerManagerImpl.class.getName()).log(Level.SEVERE, null, ex);

}

return null;

}

@Override

public synchronized void setCountOfServers(int countOfServers) { if(this.countOfServers<countOfServers)

{

for(int i=this.countOfServers; i<countOfServers; i++)

{

File file=new File("server\_classes"+fileSeparator+"server"+i); file.mkdir();

ServerInfo server=new ServerInfo(file.getAbsolutePath(),countOfMemory,i); listOfServers.add(server);

}

}else

{

while(listOfServers.get(listOfServers.size()-1).available && !listOfServers.get(listOfServers.size()).reserved

&& listOfServers.size()>countOfServers)

{

int index=listOfServers.size()-1; try{

String[] paths=listOfServers.get(index).classPath.split(pathSeparator); for(String path:paths)

{

if(path.indexOf("server\_classes"+fileSeparator+"server"+index)!=-1) delete(new File(path.substring(0, path.lastIndexOf("server"+index))));

}

}catch(Exception exc)

{

System.err.println(exc.getMessage());

exc.printStackTrace();

}

listOfServers.remove(index);

}

}

this.countOfServers=countOfServers;

}

@Override

public synchronized void setMemoryForAllServers(int countOfMemory) { for(ServerInfo info: listOfServers) info.memory=countOfMemory;

}

@Override

public synchronized boolean setMemoryForServer(int serverIdm, int countOfMemory) { try{

listOfServers.get(serverIdm).memory=countOfMemory;

}catch(Exception exc)

{

return false;

}

return true;

}

@Override

public synchronized int reserveOfServer(String host,int countOfMemory) { int index=0;

System.out.println("Менеджер серверов: Резервирование сервера"); for(ServerInfo info: listOfServers)

{

if(!info.reserved && info.available) if(index==listOfServers.size()-1

&& listOfServers.size()<countOfServers)

{ try{

String[] paths=listOfServers.get(index).classPath.split(pathSeparator); for(String path:paths)

{

if(path.indexOf("server\_classes"+fileSeparator+"server"+index)!=-1) delete(new File(path.substring(0, path.lastIndexOf("server"+index))));

}

}catch(Exception exc)

{

System.err.println(exc.getMessage());

exc.printStackTrace();

}

listOfServers.remove(index);

return -1;

}

else if(info.memory>=countOfMemory)

{

info.reserved=true; info.host=host;

System.out.println("Менеджер серверов: Зарезервирован сервер"+index); return index;

} index++;

} return -1; }

/\*\*Функция для поиска свободного порта

\* @return Возвращает свободный порт

\*/

public int getFreePort() { int freePort = 0;

try {

ServerSocket serverSocket = new ServerSocket(0);

freePort = serverSocket.getLocalPort();

serverSocket.close(); } catch (IOException ex) {

Logger.getLogger(WebConnector.class.getName()).log(Level.SEVERE, null, ex);

}

return freePort;

}

@Override

public synchronized int getMaxCountOfMemory() { int max=0; int index=0;

for(ServerInfo info: listOfServers) if(!info.reserved && info.available) if(index==listOfServers.size()-1

&& listOfServers.size()<countOfServers)

{

listOfServers.remove(index); return max;

}else if(info.memory>max) max=info.memory; return max;

}

@Override

public synchronized ServerStatus getStatusOfServer(int indexOfServer) { if(indexOfServer<listOfServers.size())

{

return listOfServers.get(indexOfServer).status;

}else return null;

}

@Override

public synchronized ArrayList<ServerStatus> getStatusesOfServers() { ArrayList<ServerStatus> listOfStatuses=new ArrayList<>();

for(ServerInfo info:listOfServers) listOfStatuses.add(info.status); return listOfStatuses;

}

@Override public synchronized ArrayList<Integer> getPortsOfServers() { ArrayList<Integer> portsOfServers=new ArrayList<>();

for(ServerInfo info:listOfServers)

{

portsOfServers.add(info.port);

}

return portsOfServers;

}

@Override

public synchronized void closeServers(ArrayList<Integer> listOfServers) { for(int index: listOfServers)

{

ServerInfo server=this.listOfServers.get(index);

server.process.destroy(); server.resetInfo();

} }

/\*\*Функция удаляет указанный файл

\* @param file - файл который необходимо удалить

\*/

private void delete(File file)

{

if(!file.exists()) return;

if(file.isDirectory())

{ for(File f : file.listFiles()) delete(f); file.delete();

} else {

file.delete();

}

}

}