Министерство цифрового развития, связи и массовых

Российской Федерации

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра «‎Информатика»

Курсовая работа по дисциплине

«‎Алгоритмы и алгоритмические языки»

На тему «Высокопроизводительные вычисления и облачные технологии»

Выполнил:

Студент группы БФИ 2102

Певин Г.И

Проверил доц. кафедры

«‎Информатика» к.п.н

Гуриков С.Р.

Москва 2022

**Оглавление**

[**ВВЕДЕНИЕ** 3](#_Toc98154797)

[**ТЕХНИЧЕСКОЕ ЗАДАНИЕ** 5](#_Toc98154798)

[**ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ** 10](#_Toc98154805)

[**1.1 Архитектура вычислительных систем** 10](#_Toc98154806)

[**1.2 Грид-системы и облачные технологии** 15](#_Toc98154809)

[**1.2.1 Концепция облачных технологий** 16](#_Toc98154812)

[**1.2.2 Сеть доставки содержимого** 20](#_Toc98154813)

[**1.2.3 Соответствие виртуальной топологии и аппаратной архитектуры** 28](#_Toc98154814)

[**1.2.4 MapReduce** 33](#_Toc98154815)

[**1.3 Распределенные файловые системы** 35](#_Toc98154816)

[**1.3.1 Google File System** 36](#_Toc98154817)

[**1.4.1** **Обзор программных интерфейсов** 45](#_Toc98154818)

# **ВВЕДЕНИЕ**

**Актуальность темы.** В настоящее время востребованность удаленного динамического доступа к услугам, вычислительным ресурсам и приложениям через интернет и просто получение информации у общества сильно возросло.

Известно**,** что облачные технологии предоставляют выгодные условия использования как для бизнеса, так и для рядового пользователя. Компании, предоставляющие облачные решения, используют слоган: «плати только за то, что используешь».

Несомненно, это новая технология, которая быстро развивается. На данный момент многие бояться именно самого факта, что информацию будут хранить сторонние люди. Но это позволяет нам получать большие вычислительные мощности, информацию из любой точки мира. Так же сказывается недостаточное на данный период времени качество, стабильность и скорость интернет - соединений, что создает ощутимые трудности.

Можно сказать,что это несомненно новая технология, которая только начала входить в использование. У нее есть свои плюсы: экономия для потребителей, борьба с пиратством для разработчиков, минимизация затрат в IT сфере для бизнеса, самообслуживание по требованию, быстрая эластичность. Все перечисленное являются, несомненно актуальным в наше время, так как с развитием технологий появляется необходимость в более мощных вычислительных возможностях, что так же подтверждает актуальность темы курсовой работы. Дополнительным подтверждением актуальности является то, что в современном мире все ищут упрощения, что и дают облачные технологии.

**Объектом исследования**, проведенного в рамках данной курсовой работы, являются облачные технологий и высокопроизводительные вычисления.

**Предметом исследования** являются классификации облачных сервисов.

**Цели работы и задачи исследования.** Цель работы – исследовать основы высокопроизводительных приложений. Поставленная цель определила следующие основные задачи исследования:

1. Показать, в чем отличие классификаций облачных сервисов;
2. Анализ перспективы развития облачных технологий;
3. Разработать тестовую программу, которая использует многопоточность процессора.

**Методы исследования.** В ходе выполнения данной курсовой работы был использован теоретический метод исследования, позволяющий более явно продемонстрировать актуальность темы. Фундаментом теоретического метода исследования послужила работа Дружинина Д.В.

# **ТЕХНИЧЕСКОЕ ЗАДАНИЕ**

1. **Введение**

Областью применения разрабатываемого программного продукта является углубленное изучение, а также контроль качества усвоения студентами основных определений, принципов и особенностей, используемых в бизнесе, образовании и в госсекторе. Они нужны практически любой компании и организации, работающей с данными и имеющей свою ИТ-инфраструктуру, так как делают бизнес более гибким, быстрым и эффективным, а значит более конкурентоспособным.

2. **Основание для разработки**

Основание для разработки является задание, выполненное в соответствии с заданием, полученным от кафедры «Информатика» Московского технического университета связи и информатики и утвержденное научным руководителем доцентом кафедры «Информатика» к.п.н. Гуриковым С.Р. от 05.02.2022.

Наименование разработки: Высокопроизводительные технологии и облачные технологии.

1. **Назначение разработки**

Программный продукт предназначен для ознакомления с основными теоретическими положениями по теме курсовой работы и проверки знаний студента с помощью тестовой программы.

4. **Требования к программе или программному изделию**
   1. **Требования к функциональным характеристикам**

Разработанный программный продукт должен обеспечить выполнение следующих функций:

* Обеспечить высокопроизводительную и стабильную программу
* Вывести пользователю конечную программу;
* Сформулировать интуитивно понятный интерфейс

Программная разработка должна быть написана в четком соответствии и материалами занятий по дисциплине «Алгоритмы и алгоритмические языки». Структуру программного кода тестовых вопросов изменять нельзя, она должна соответствовать материалам занятий.

Студенту, за счет часов самостоятельной работы разрешается заниматься дополнительным поиском информации с целью расширения возможности своей программной разработки, с последующим описанием их в пояснительной записке.

1. 2. **Требования к надежности**

Разрабатываемое программное обеспечение должно быть спроектировано таким образом, чтобы обеспечить защиту и надежную работу при наличии ошибок во входных данных и/или от некорректных действий пользователя - предполагается, что программный продукт должен быть спроектирован таким образом, чтобы внутренняя или внешняя (некритическая для системы) ошибка не приводила к аварийной остановке.

* 1. **Условия эксплуатации**

Климатические условия эксплуатации, при которых должны обеспечиваться заданные характеристики, должны удовлетворять требованиям, предъявляемым к техническим средствам в части условий их эксплуатации.

Программа будет работать в температурном режиме от + 5 до + 35 °C при относительной влажности 90 % и атмосферном давлении 462 мм.рт.ст., поскольку такие условия приблизительно соответствуют условиям эксплуатации современных компьютеров непромышленного исполнения.

* 1. **Требования к составу и параметрам технических средств**

Для корректной работы программного продукта вычислительная система должна обладать следующими характеристиками:

Минимальные системные требования:

Видеокарта: 512MB (DirectX 10, 10.1, 11)

Процессор: 1.80 GHz

Оперативная память (ОЗУ): 2GB

Свободного места на диске: 800MB

Периферийные устройства: клавиатура, мышь;

Для корректной работы вычислительной среды необходимо наличие системного программного обеспечения, основным элементом которого является операционная система. В связи с тем, что Microsoft является лидером в сегменте продаж операционных систем, то целесообразнее использовать операционную систему семейства Windows не ниже Windows 8.

* 1. **Требования к информационной и программной совместимости**

Требования к информационным структурам (файлов) на входе и выходе не предъявляются.

Исходные коды программы должны быть реализованы на языке Visual С++. В качестве интегрированной среды разработки программы должна быть использована среда MS Visual Studio.

Системные программные средства, используемые программой, должны быть представлены лицензионной локализованной версией операционной системы.

* 1. **Требования к маркировке и упаковке**

Программа поставляется в виде программного изделия - внешнем флеш- носителе.

Программное изделие должно иметь маркировку с обозначением товарного знака компании-разработчика, типа (наименования), номера версии, порядкового номера, даты изготовления и номера сертификата соответствия Госстандарта России (если таковой имеется). Маркировка должна быть нанесена на программное изделие в виде наклейки, выполненной полиграфическим способом с учетом требований ГОСТ 9181-74.

Упаковка флеш-носителя - пакет для хранения.

Упаковка программного изделия должна проводиться в закрытых вентилируемых помещениях при температуре от плюс 15 до плюс 40 °С и относительной влажности не более 80 % при отсутствии агрессивных примесей в окружающей среде.

Подготовленные к упаковке программные изделия укладывают в тару, представляющую собой коробки из картона. Для заполнения свободного пространства в упаковочную тару укладываются прокладки из гофрированного картона или пенопласта. На верхний слой прокладочного материала укладывается товаросопроводительная документация - упаковочный лист и ведомость упаковки.

* 1. **Требования к транспортированию и хранению**

Транспортировка разрабатываемого программного продукта должна осуществляться студентом в назначенный срок по указанию преподавателя, принимающего результаты курсовой работы; Ответственным за хранение программного продукта является студент; Результаты выполненной работы хранятся на кафедре «Информатика» в течение года.

1. **Требования к программной документации**

Предварительный перечень программной документации:

Пояснительная записка к курсовой работе, оформленная в соответствии с ГОСТ 7.32-2001);

1. **Технико-экономические показатели**

В данной работе не рассчитываются.

1. **Стадии и этапы разработки**

Стадии и этапы разработки определены в план-графике выполнения курсовой работы

1. **Порядок контроля и приемки**

— Контроль и приемка программного продукта осуществляется в течение семестра поэтапно в соответствии с план-графиком;

— Курсовая работа подлежит защите, в ходе которой студент представляет свою разработку;

— В случае невыполнения план-графика, студенту могут быть выданы дополнительные задания;

— В случае регулярного невыполнения план-графика, студенту может быть выдано задание на защиту курсовой работы, в ходе выполнения которого он обязан продемонстрировать свои практические навыки;

— Оценку программного продукта производят преподаватели, осуществляющие проверку курсовых работ.

**ГЛАВА 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

## **1.1 Архитектура вычислительных систем**

**Классификация вычислительных систем.** Рассмотрим классификацию вычислительных систем Флинна, основанную на анализе количества потоков команд и количества потоков данных.

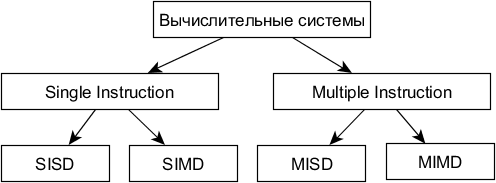


Рисунок. 1.1 Классификация вычислительных систем Флинна

SISD (англ. Single Instruction, Single Data) – системы с одним потоком команд и одним потоком данных. Классическим примером является архитектура фон Неймана со строго последовательным выполнением команд.

SIMD (англ. Single Instruction, Multiple Data) – системы с одним потоком команд и множеством потоков данных. Данная архитектура предполагает наличие управляющего устройства (контроллера) и набора процессорных элементов. Управляющий модуль принимает, анализирует и выполняет команды. Если в команде встречаются данные, контроллер рассылает на все процессорные элементы команду, и эта команда выполняется на нескольких или на всех процессорных элементах. Каждый процессорный элемент имеет свою собственную память для хранения данных. К этому типу систем относятся векторные компьютеры, в которых операндами некоторых команд могут выступать одномерные массивы данных (векторы).

MISD (англ. Multiple Instruction, Single Data) – системы с множеством потоков команд и одним потоком данных. Самая простая и распространённая система этого класса – локальная вычислительная сеть (ЛВС) персональных компьютеров, работающая с единой базой данных.

MIMD (англ. Multiple Instruction, Multiple Data) – системы с множеством потоков команд и множеством потоков данных. К этому классу относится подавляющее большинство современных высокопроизводительных вычислительных систем. Рассмотрим классификацию таких систем.

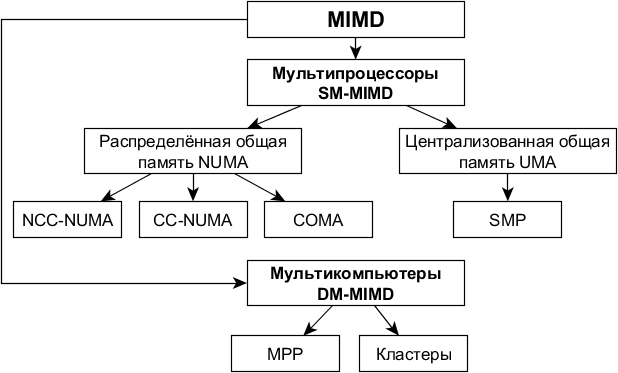
Рассмотрим Классификации MIMD систем.

Рисунок 1.1 - Классификация MIMD систем

SM-MIMD (англ. Shared Memory Multiple Instruction, Multiple Data) – системы с разделяемой памятью (мультипроцессоры).

DM-MIMD (англ. Distributed Memory Multiple Instruction, Multi- ple Data) – системы с распределённой памятью (мультикомпьютеры), в которых один процессор не может обратиться к памяти любого другого процессора.

Мультипроцессоры разделяют на вычислительные системы с централизованной общей памятью (англ. UMA, Uniform Memory Access) и вычислительные системы с распределённой общей памятью (NUMA, Non-Uniform Memory Access). При первом подходе обеспечивается однородный доступ к памяти, обращение любого процессора к любой ячейке памяти занимает одинаковое время. При втором подходе память физически распределена между процессорами или их группами и, как следствие, время доступа процессора к различным ячейкам памяти может существенно разниться.

Одной из основных проблем, возникающих при организации вычислений на мультипроцессорах, является проблема обеспечения однозначности (когерентности) данных, находящихся в кэш- памяти различных процессоров. Различные типы мультипроцессорных систем используют различные подходы к обеспечению когерентности данных в кэшах, входящих в состав такой системы процессоров.

Симметричная многопроцессорная архитектура (англ. SMP, Symmetric Multiprocessor) предполагает наличие в системе набора однородных процессоров и массива общей памяти. Когерентность данных в кэшах различных процессоров обеспечивается на аппаратном уровне. Наличие централизованной общей памяти упрощает взаимодействие процессоров между собой, но такая архитектура является ограниченно масштабируемой из-за требования малых задержек когерентных связей. Реальные SMP-системы содержат не более 32 процессоров.

Архитектура CC-NUMA (англ. Cache Coherent Non-Uniform Memory Access) представляет собой совокупность SMP-узлов с собственной памятью, объединённых между собой общей шиной или коммутатором. Так формируется физически распределённая память с единым адресным пространством. Каждый SMP-узел имеет собственную локальную память и может устанавливать статические и динамические соединения с модулями памяти других узлов. В архитектуре CC-NUMA используется аппаратнореализованная кэш-когерентность. Такой подход позволяет объединить до 252 процессоров и сохранить неизменность программного обеспечения SMP-приложений.

Альтернативой архитектуре CC-NUMA является COMA (англ. Cache-Only Memory Architecture). COMA-узел не имеет собственной оперативной памяти (ОП), что компенсируется наличием большого количества кэш-памяти. В COMA отсутствует иерархия памяти, а адресное пространство составлено из адресных пространств всех кэшей. Фрагменты данных не привязаны статически к определённому модулю памяти и не имеют уникального адреса, остающегося неизменным в течение всего времени существования переменной. При работе с фрагментами данных используются две стратегии: копирование и миграция. Копирование применяется при обращении к фрагменту данных нескольких процессоров на чтение, а миграция – при модификации фрагмента. В зависимости от того, преобладает в приложении копирование или миграция, более эффективной оказывается либо архитектура CC-NUMA, либо COMA. Так, CC-NUMA демонстрирует лучшие показатели производительности при сравнительно частой модификации фрагментов, используемых различными процессорами, так как при этом не требуется обновлять таблицу расположения фрагментов данных, как в COMA. COMA более эффективна при редком осуществлении модификации совместно используемых фрагментов за счёт того, что размер кэш-памяти в COMA-системах больше, чем в CC-NUMA и, как следствие, выше вероятность наличия копии такого фрагмента на каждом процессоре.

Архитектура NCC-NUMA (англ. Non-Cache Coherent Non- Uniform Memory Access) отличается от CC-NUMA отсутствием поддержки кэш-когерентности на аппаратном уровне. Такие системы лучше масштабируются и, как правило, имеют меньшую стоимость, чем CC-NUMA, но требуют использования дополнительных средств для достижения кэш-когерентности.

Вычислители с массовым параллелизмом (англ. MPP, Massively Parallel Processor) характеризуются отсутствием общей памяти.

Узлы, состоящие, как правило, из процессора, памяти и устройств ввода-вывода, объединяются между собой специализированным соединением. На каждом узле имеется своя операционная система (ОС) и уникальное адресное пространство физической памяти. Для обмена данными между узлами используются программные средства, например, обмен сообщениями. Такая архитектура хорошо масштабируется, но требует эффективного распределения данных между узлами для минимизации обмена данными между ними, так как получение данных от другого узла является потенциально дли тельной операцией. Так получение данных от другого узла занимает не менее чем в 100 раз больше времени, чем при архитектуре CC-NUMA, где есть аппаратная поддержка кэш-когерентности.

Кластер. Стоит сразу оговориться, что здесь под кластером будем понимать кластерный вычислитель, в то время как существуют также кластеры приложений, цель создания которых состоит в повышении надёжности и отказоустойчивости серверов приложений.

Кластер отличается от вычислителей с массовым параллелизмом гетерогенной структурой. Это означает, что узлы кластера могут отличаться на аппаратном уровне и иметь различные ОС. Существует несколько способов построения кластера. Так кластер может на аппаратном уровне представлять собой набор персональных компьютеров (ПК), объединённых в локальную сеть или подключённых к сети Интернет. При таком способе построения стоимость кластера минимальна, но низкая производительность коммуникационной подсистемы и большое количество низкопроизводительных узлов требуют разбиения задачи на максимальное количество подзадач и накладывают жёсткие ограничения на объём данных, пересылаемых между узлами.

В другом варианте компоновки кластера каждый узел является мультипроцессором, при этом возможно построение кластеров, разные узлы которых имеют различную архитектуру (SMP, CC-NUMA и т.д.). При таком варианте компоновки кластера достаточно иметь значительно меньшее количество узлов в сравнении с кластером, узлами которого являются ПК. Преимущество такого типа кластера состоит в том, что достаточно разбить задачу на сравнительно крупные подзадачи за счёт высокой производительности отдельных узлов. Возможно построение кластеров, включающих в качестве узлов как мультипроцессоры, так и ПК.



**1.2 Грид-системы и облачные технологии**

**Понятия и свойства грид-систем.** Грид-система (англ. grid) представляет собой программно- аппаратный комплекс, построенный на основе кластерного вычислителя. Узлами такого кластера обычно являются ПК. Программное обеспечение грид-систем позволяет выполнять вычисления в низкоприоритетном режиме во время простоя компьютера-узла, а в остальное время он может выполнять функции ПК пользователя.

Грид-системы имеют высокоуровневый интерфейс, который позволяет предоставлять пользователю прозрачный интерфейс к своим ресурсам, таким как процессорное время, данные, программное обеспечение. Пользователь взаимодействует с грид-системой, как с одним, но гораздо более мощным компьютером. Грид-системы ещё называют метакомпьютерами или «виртуальнымм суперкомпьютерами». Состав грид-системы может меняться динамически, в ходе выполнения расчётов. Так одни узлы могут подключаться к ней, другие выходить из её состава.

Наилучшим образом для решения с помощью грид-систем подходят задачи переборного и поискового типа, где вычислительные узлы практически не взаимодействуют друг с другом, функционируя большую часть времени в автономном режиме.

С точки зрения выделения вычислительных ресурсов грид- системы классифицируют следующим образом:

1. Добровольные.
2. Научные.
3. Коммерческие.

В добровольных грид-системах, для участия в которых любой желающий может предоставить вычислительные ресурсы своего ПК, появляются дополнительные риски. Эти риски связаны в первую очередь с возможностью умышленного предоставления отдельными узлами грид-системы ошибочных или искажённых результатов вычислений. Для снижения подобных рисков грид- система выдаёт одно и то же задание как минимум двум узлам из различных сегментов сети. Результат считается достоверным, если оба узла вернули идентичный результат. Конечно, такой подход не даёт стопроцентной гарантии отсутствия искажённых результатов в результирующем наборе данных.

Научные грид-системы объединяют вычислительные ресурсы научных организаций. Коммерческие грид-системы предполагают предоставление вычислительных ресурсов клиенту в форме платного сервиса.

Одной из наиболее известных и самой производительной на момент написания данного пособия грид-системой является Berke- ley Open Infrastructure for Network Computing (BOINC). Эта грид-система имеет среднюю производительность более 130 терафлопс, в проекте зарегистрировано более 3 млн. участников. BOINC относится к добровольным грид-системам и призвана выполнять вычисления в различных научных задачах, таких как изучение процессов глобального потепления, обнаружение космических объектов и др.



### **1.2.1 Концепция облачных технологий**

Суть облачных технологий (облачных вычислений) состоит в предоставлении программных и виртуализированных аппаратных ресурсов в качестве сервиса. Облачная обработка данных предполагает постоянное хранение данных на серверах и их временное кэширование на клиентской стороне. Рассмотрим основные свойства облачных технологий.

*Самообслуживание по требованию*. Потребитель может самостоятельно изменять объём используемых ресурсов без взаимодействия с представителями поставщика услуг.

*Универсальный доступ по сети*. Услуги доступны потребителям по сети передачи данных вне зависимости от используемого терминального устройства.

*Объединение ресурсов*. Поставщик услуг объединяет ресурсы для обслуживания большого числа потребителей в единый пул для динамического перераспределения этих ресурсов между потребителями.

*Быстрая эластичность*. Объём предоставляемых ресурсов может быть существенно изменён в любой момент как в сторону увеличения, так и в сторону уменьшения. С точки зрения потребителя предоставляемые ресурсы выглядят неограниченными.

*Учёт потребления*. Поставщик ведёт учёт потребления ресурсов на определённом уровне абстракции (например, объём хранимых данных, пропускная способность, количество пользователей, количество транзакций). Эти данные доступны для поставщика и потребителя услуг.

Облачные технологии получили широкое распространение в первую очередь потому, что для многих рядовых пользователей и представителей среднего и малого бизнеса оказалось дешевле платить за время или объём использования программных и виртуализированных аппаратных ресурсов, чем покупать лицензированное ПО или аппаратное обеспечение.

Рассмотрим **классификацию** облачных сервисов по типу ресурса, к которому предоставляется доступ:

* SaaS (Software as a Service).
* PaaS (Platform as a Service).
* IaaS (Infrastructure as a Service).
* DaaS (Desktop as a Service, Data as a Service).
* CaaS (Communications as a Service).

SaaS (Приложение как сервис) – вариант, при котором пользователю предоставляется ПО по подписке. SaaS предназначен для конечного пользователя. В качестве примера можно привести Google Apps, Microsoft Cloud Service.

PaaS (Платформа как сервис). В облаке функционирует набор программ, сервисов и библиотек, на основе которых предлагается разрабатывать свои приложения. Таким образом, PaaS предназначен для разработчиков ПО. Примерами PaaS являются Google Apps Engine, Microsoft Azure.

IaaS (Инфраструктура как сервис). Клиент получает в своё распоряжение виртуализированные аппаратные ресурсы, то есть клиенту предоставляются не конкретное аппаратное обеспечение, а такие ресурсы как дисковое пространство, процессорное время в эквиваленте какого-либо реального процессора, пропускная способность. Преимущество Iaas по сравнению с покупкой конкретного аппаратного обеспечения для клиента состоит в возможности динамического масштабирования. Например, небольшая компания начинает продажи нового приложения, имеющего клиент- серверную архитектуру. Заранее неизвестно, как быстро будет увеличиваться количество пользователей. Поэтому сервер с низкой производительностью может не справиться с резко возросшей нагрузкой, а покупка дорогого производительного сервера до начала массовых продаж может быть компании не по карману. В случае использования виртуального сервера компания получает возможность оплачивать только использованное процессорное время и сетевой трафик. Поэтому при малом количестве пользователей затраты на содержание виртуального сервера будут невелики, а в случае резкого роста количества пользователей, проблемы масштабирования провайдер IaaS берёт на себя. Примерами IaaS являются Amazon Elastic Compute Cloud, OpenNebula.

Сокращение DaaS имеет два различных значения применительно к облачным технологиям.

DaaS (Desktop as a Service, Рабочее место как сервис). Отличие от SaaS здесь состоит в том, что пользователю предоставляется не отдельная программа, а виртуальное рабочее место – программный комплекс, включающий ОС и прикладные приложения, необходимые для работы. Примером Desktop as a Service является Citrix XenDesktop.

DaaS (Data as a Service, Данные как сервис). В качестве сервиса пользователю предоставляется доступ к данным провайдера, полученным из различных каналов с различных устройств, это могут быть известные и анонимные данные. Помимо исходных данных пользователю предоставляются средства их анализа и агрегирования. Примером Data as a Service является Oracle Cloud.

CaaS (Коммуникации как сервис). В качестве сервиса пользователю предоставляются услуги связи. Это в первую очередь IP- телефония, сервисы видеоконференций и сервисы мгновенных сообщений. В качестве примера можно привести Google Hangouts. Для использования этого сервиса пользователю не нужно устанавливать на свой компьютер никаких приложений, достаточно зайти на веб-сайт Google Hangouts.

Стоит отметить, что созвучный рассмотренным выше термин HaaS (Hardware as a Service, Аппаратное обеспечение как сервис) не относится к сфере облачных технологий. HaaS предполагает физическую передачу клиенту аппаратного обеспечения на определённый срок.

Можно выделить следующие **модели развёртывания** облачных систем:

* Частное облако.
* Публичное облако.
* Гибридное облако.

Частное облако предназначено для использования одной организацией, включающей несколько потребителей. В качестве потребителей могут выступать подразделения этой организации, её клиенты или подрядчики.

Публичное облако предназначено для использования широким кругом лиц. Публичные облака могут быть бесплатными для поль зователей.

Гибридное облако – комбинация из нескольких различных облачных инфраструктур, остающихся уникальными объектами, но связанных между собой технологиями передачи данных. В качестве примера можно привести кратковременное использование ресурсов публичных облаков для балансировки нагрузки между облаками. Гибридное облако также создаётся в случае, когда приватные данные пользователя хранятся в частном облаке, но для расширения его функциональных возможностей используются приложения из публичного облака.

### **1.2.2 Сеть доставки содержимого**

Рассмотрим распространённый тип сетевых сервисов, базирующихся на грид-системах – сеть доставки содержимого (англ. Con- tent Delivery Network, CDN). CDN – это географически распределённая сеть прокси-серверов, создаваемая для повышения доступности(в смысле процента времени, когда ресурс доступен) и производительности. В CDN экземпляры сервиса размещаются поблизости от областей концентрации конечных пользователей, что позволяет также уменьшить время отклика.

В технологии CDN запрос к сервису переадресуется к узлу, оптимальному по некоторому критерию, например обеспечивающий минимальное количество. При выборе узла также может учитываться его уровень доступности и текущей нагрузки.

Одной из распространённых разновидностей CDN является *веб- кэш*. Веб-кэш хранит популярный контент на тех серверах, на которых потребность в нём максимальна. Существует два способа наполнения веб-кэша – при поступлении запроса от клиента и предзагрузка с других узлов. В первом случае данные загружаются на прокси-сервер при первом запросе от конечного пользователя, что позволяет ускорить обработку следующего запроса на получение тех же данных. Во втором случае при вводе очередного прокси-сервера в эксплуатацию на него сразу загружаются популярные данные в соответствии с ранее набранной статистикой, чтобы уже первый запрос к ним обрабатывался быстрее. Описанные способы наполнения веб-кэша не являются взаимоисключающими.

**Обнаружение ресурсов в грид-системах.** В больших грид-системах возникает проблема обнаружения ресурса, максимально соответствующего по характеристикам запросу. Выбор может быть продиктован целым рядом факторов, в зависимости от характеристик приложения. Так, приложения, выполняющие хорошо поддающиеся распараллеливанию трудоёмкие вычисления, будут предъявлять требования в основном к производительности ЦП, объёму и быстродействию ОП и накопителя на узлах-кандидатах.

Для приложений, активно использующих сетевую инфраструктуру, таких как CDN и мониторинговые приложения, обеспечивающие безопасность, может оказаться важнее расположение экземпляров сервисов в заданных сетевых локациях – возле потенциальных клиентов или в хорошо распределённых локациях в топологии сети. Также важно, чтобы узлы, на которых будут размещены сервисы, были соединены между собой каналами связи с низкими задержками и высокой пропускной способностью.

И, наконец, гибридным приложениям, таким как многопользовательские игры, могут требоваться оба типа атрибутов узла, например, низкая нагрузка при обсчёте логики игры и низкая задержка для пользователей для хорошей интерактивной производительности.

К современным системам обнаружения ресурсов предъявляются следующие требования:

1. Высокая масштабируемость. Система должна быть способна работать с десятками тысяч узлов.
2. Высокая доступность. Система должна быть защищённой от сбоев, быстро восстанавливаться от них в автоматическом режиме. Это требование продиктовано тем фактом, что при поступлении новых запросов распределённая инфраструктура становится фактически недоступной, если недоступна система обнаружения ресурсов.
3. Учёт как статических (редко изменяемых), так и динамических характеристик узлов. К статическим характеристикам относится установленная на узле ОС, доступные лицензии ПО, расположение узла. К динамическим характеристикам относится количество доступных ресурсов, таких как процессорное время, объём ОП и дисковой памяти.

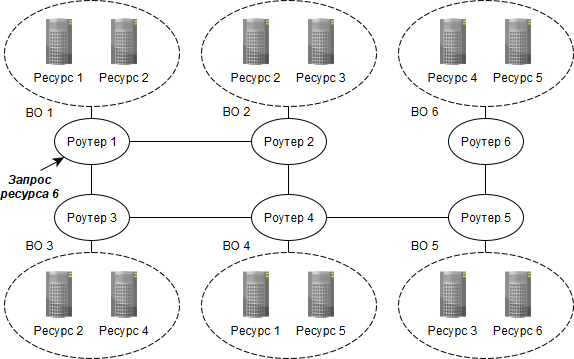
Если для решения этой проблемы использовать единственный управляющий узел, то с ростом грид-системы, с повышением степени её территориальной распределённости, он быстро становится фактором, повышающим среднее время отклика и ограничивающим суммарную производительность системы.

Поэтому набирают популярность децентрализованные подходы к обнаружению ресурсов, которые предполагают наличие множества управляющих узлов в грид-системе, каждый из которых обрабатывает запрос на обнаружение ресурсов в ближайших сегментах сети. Стоит отметить, что управляющие узлы не обязательно должны быть обособлены, их роль могут выполнять и вычислительные узлы сети. Рассмотрим некоторые децентрализованные подходы к обнаружению ресурсов.

Подход, основанный на взаимодействии виртуальных организаций. *Виртуальная организация* (*ВО*) – это временный или постоянный набор географически распределённых вычислителей, соединённых с помощью коммуникационной подсистемы для выполнения поставленной задачи. В рамках этого подхода каждая ВО содержит набор вычислительных узлов (ресурсов), а также один роутер, посредством которого осуществляется взаимодействие с другими ВО в системе. Каждый роутер ВО ведёт таблицу маршрутизации, составленную для известных ему ресурсов. Для каждого типа ресурса роутер *Ri* хранит данные о пути минимальной длины, измеряемой в количестве переходов от *Ri* до известного ему ресурса этого типа.

Таблица маршрутизации ВО содержит три раздела: локальные ресурсы, ресурсы соседних ВО и удалённые ресурсы. К локальным ресурсам относятся ресурсы, входящие в состав ВО. Соседними для ВО1 считаются такие ВО, роутер которых напрямую соединён с роутером ВО1, прочие ВО считаются удалёнными. В разделе, содержащем данные об удалённых ресурсах, дополнительно хранится информация о расстоянии до них, измеряемом в количестве переходов от роутера ВО до удалённого ресурса.

Рассмотрим пример обработки запроса на получение ресурса в грид-системе. ВО1 получает запрос на ресурс 6-го типа. Такого ресурса нет в ВО1, а также в соседних ВО2 и ВО3, информация о ресурсах, которых также хранится в таблице маршрутизации ВО1, поэтому ВО1 перенаправляет запрос в ВО2 и ВО3. Так как в этих ВО, а также в соседних с ними ВО ресурс запрошенного типа отсутствует, ВО1 увеличивает радиус запроса на 1. Теперь запрос передаётся роутеру ВО4, в таблице маршрутизации которого содержится информация о том, что ресурс 6-го типа находится в соседнем с ВО4 ВО5 на расстоянии 2.

Рисунок. 1.2.2 - Пример обработки запроса на получение ресурса в грид - системе при подходе, основанном на ВО

Таким образом, для обнаружения ресурса 6-го типа понадобился поиск на расстоянии всего двух переходов от исходного роутера ВО1. После обнаружения искомого ресурса ответ от роутера ВО4 проходит через все участвовавшие в поиске роутеры, при этом обновляются их таблицы маршрутизации. В следующий раз при поступлении запроса на ресурс 6-го типа, адресованный ВО1, ВО2 или ВО3, запрос будет сразу перенаправлен в ВО5

Scalable Wide-Area Overlay-Based Resources Discovery (SWORD). Пользователю необходимо описать требования к инфраструктуре, на которой он собирается развернуть свои приложения, используя специальный язык запросов, основанный на нотации XML. Например, сервис, предоставляющий доступ к потоковым медиа данным, может запросить несколько виртуальных кластеров, каждый из которых должен располагаться поблизости от крупной группы пользователей этого сервиса. Каждый такой кластер рассматривается как *класс эквивалентности*. На практике это означает, что на определённом этапе обработки запроса система обнаружения ресурсов для каждого такого кластера подберёт набор эквивалентных кандидатов. На следующих этапах будет выбран кластер-кандидат в соответствии с дополнительными критериями. Каждый кластер будет состоять из узлов, суммарно обеспечивающих достаточно дискового пространства для хранения медиа-файлов, которые должны быть доступны в заданном регионе, а также каждый кластер должен иметь как минимум одно соединение достаточной пропускной способности хотя бы с одним из сайтов, генерирующих контент, чтобы новый контент мог быть быстро распределён по кластерами.

Пользователи SWORD определяют набор обязательных и желательных значений для характеристик как отдельных узлов, так и межузловых соединений. SWORD позволяет задать штраф (стоимость), который применяется, когда выбираются узлы, отвечающие обязательным требованиям, но не отвечающие более жёстким желательным требованиям. Например, в качестве желательной скорости отправки данных для всех узлов приложения указывается 1 Мб/с, что соответствует нулевой стоимости. Пропускная способность, превышающая 512 Кб/с может быть достаточной, но ей соответствует ненулевая стоимость, в то время как пропускная способность ниже 512 Кб/с является недостаточной для приложения.

SWORD пытается найти конфигурацию с минимальной стоимостью, отвечающую пользовательским требованиям.

В языке описания запросов SWORD вводится понятие *группы узлов*, для которой задаётся количество составляющих её узлов, а также требования, предъявляемые к каждому такому узлу, что позволяет описывать виртуальные кластеры. Язык запросов SWORD позволяет также задавать требования к каналам связи между членами различных групп, к примеру, чтобы в каждой группе был как минимум один узел, имеющий канал связи с задержкой менее 100 мс хотя бы с одним из узлов другой группы.

**Архитектура**. Рассмотрим высокоуровневую архитектуру SWORD. Пользователь пишет XML-запрос (шаг 1) и отправляет его любому управляющему узлу SWORD (шаг 2). Запрос передаётся распределённому обработчику запросов – программному компоненту, запущенному на этом узле. Обработчик запросов опрашивает ряд узлов в диапазоне, соответствующем поступившему запросу (шаг 3). Как только распределённый запрос завершён (шаг 4), список узлов-кандидатов, а также соответствующие им характеристики передаются оптимизатору, запущенному на том же узле, который получил исходный запрос от пользователя (шаг 5). Оптимизатор выбирает подмножество узлов, суммарно минимизирующее стоимость конфигурации, и возвращает его пользователю (шаги 6 и 7).

Как видно из описания высокоуровневой архитектуры SWORD, в этой системе реализована распределённая схема обнаружения ресурсов, предполагающая наличие множества управляющих узлов, число которых растёт с увеличением размера и степени географической распределённости грид-системы. Данные о вычислительных узлах продублированы на нескольких управляющих узлах, чтобы исключить ситуацию, когда сегмент сети становится де факто недоступным при выходе из строя единственного управляющего узла. Стоит отметить, что управляющие и вычислительные узлы здесь – скорее логические понятия, так как один физический узел грид-системы может совмещать обе роли.

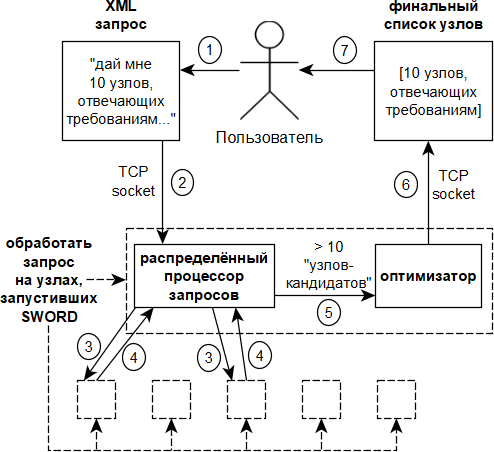


Рисунок 1.2.2 - Архитектура SWORD

В общем случае, SWORD находит субоптимальное решение, так как накладываются ограничения как на количество опрашиваемых в ходе обработки распределённого запроса узлов, так и на максимальное время работы оптимизатора. Эти значения могут быть заданы в самом запросе.

Задача балансировки нагрузки на узлы в SWORD решается путём включения текущего уровня нагрузки в список критериев при выборе узла в ходе выполнения запроса.

Таким образом, SWORD за счёт поддержки языка запросов, обладающего высокой выразительностью, предоставляет пользователю гораздо более обширные возможности нежели подход, основанный на взаимодействии виртуальных организаций, рассмотренный в предыдущем пункте.

### **1.2.3 Соответствие виртуальной топологии и аппаратной архитектуры**

Распределённые приложения состоят из набора сравнительно независимых *задач* (*потоков выполнения*), которые могут быть запущены на разных узлах вычислительной сети. В этом разделе будут употребляться оба термина – задача и поток выполнения, в качестве синонимов.

В ходе организации высокопроизводительных вычислений возникает проблема *расположения* (в англоязычной литературе locality) сразу на нескольких уровнях:

* выбор узла сети для конкретного *потока выполнения* приложения (процесса или потока);
* выбор места хранения данных потока выполнения. Это может быть локальный диск, ОП на узле потока выполнения либо отдельный узел сети;
* выбор ядра процессора для запуска потока выполнения (ввиду иерархической организации памяти).

Так как время передачи данных между потоками выполнения зависит от их взаимного расположения, его оптимизация способна существенно повысить производительность вычислительной системы. Это задача *сопоставления топологий*, в ходе решения которой устанавливается соответствие между узлами вычислительной сети и потоками выполнения приложения. Для решения этой задачи требуется определить *виртуальную топологию* приложения, которая отражает взаимосвязи между потоками выполнения.

Стоит отметить, что проблема расположения данных практически не учитывается в таких распространённых технологиях параллельных вычислений как OpenMP и MPI. Так OpenMP использует глобальное адресное пространство, а также секционированное глобальное адресное пространство (англ. PGAS), не учитывая иерархическую организацию памяти. MPI в свою очередь не учитывает топологию сети, всегда рассматривая её как плоскую, где все узлы подключены к одному маршрутизатору.

Виртуальная топология может быть *статической* или *динамической*. В первом случае количество потоков выполнения и набор взаимосвязей между ними не меняются в ходе функционирования приложения. Если существует возможность изменения хотя бы одного из этих параметров, виртуальная топология считается динамической. Например, в приложениях, построенных на базе OpenMP, используется динамическая топология, так как потоки могут создаваться и уничтожаться при входе и выходе из параллельных секций.

Чтобы подойти к задаче сопоставления топологий, необходимо не только понимать, какие статические и динамические связи присутствуют между потоками выполнения, но и собрать данные об аппаратной платформе. Такие данные включают топологию вычислительной сети, особенности иерархической организации памяти её узлов, в том числе наличие NUMA-эффектов.

Задача сопоставления топологий имеет два варианта постановки. В первом традиционном варианте требуется всем потокам выполнения сопоставить узлы вычислительной сети. Во втором вари анте требуется настроить вычислительную сеть таким образом, чтобы она максимально соответствовала виртуальной топологии. Такая постановка задачи становится возможной при использовании, например, программно-определяемой сети (англ. software- defined networking, SDN).

**Формализация задачи**. Задача сопоставления топологий может быть формализована как задача минимизации различных метрик. При решении этой задачи сеть обычно моделируется взвешенным графом, где ∈ ℕ является множеством вычислительных узлов, ω𝐻 (𝑢, 𝑣) ∈ ℝ, 𝑢, 𝑣 ∈ 𝑉𝐻 задаёт вес ребра между точками *u* и *v*. Функция *RH*(*u*, *v*) задаёт маршрут как распределение вероятностей на множестве путей *P*(*u*, *v*) между точками *u* и *v*. В упрощённых моделях функция *RH* может игнорироваться, а ω*H* может быть заменено бинарными значениями, обозначающими наличие либо отсутствие маршрута между точками *u* и *v*.

Виртуальная топология приложения часто моделируется взвешенным графом *A* = (***VA***, ω*A*), где ***VA*** – это множество потоков выполнения, ω*A*(*u, v*) задаёт объём взаимодействия между ними, выраженный посредством некоторой метрики. В качестве такой метрики может использоваться суммарный объём переданных между потоками выполнения данных за заданную стадию выполнения приложения, либо количество переданных сообщений.

Сопоставление топологий предполагает наличие отображения σ: ***VA*** → ***VH***, где σ сопоставляет каждому потоку выполнения 𝑠 ∈ 𝑉Æ целевой вычислительный узел 𝑡 ∈ 𝑉𝐻. σ, в общем случае, задаёт отображение многие-ко-многим, т.е. возможно как сопоставление потоку выполнения из виртуальной топологии приложения нескольких вычислительных узлов, так и одновременное функционирование нескольких потоков выполнения на одном вычислительном узле.

Для возможных вариантов отображения σ рассчитываются различные метрики, для которых затем решается задача минимизации для нахождения оптимального или субоптимального отображения σ'. Рассмотрим две такие метрики – *расширение* (англ. dilation) и *перегруженность* (англ. congestion). Расширение неформально задаётся как максимум или сумма попарных дистанций взаимодействующих потоков выполнения из ***A***, отображённых в ***H***. Например, пусть *dH*(*x*, *y*) задаёт кратчайшее расстояние между точками 𝑥, 𝑦 ∈ 𝑉𝐻; метрика расширения в виде взвешенной суммы определяется как:

Метрика расширения позволяет оценить, какое количество раз пакеты будут переданы по сети в ходе выполнения распределённого приложения. В случае, когда задана функция маршрутизации *RH*, её использование вместо *dH* (*x*, *y*) даёт возможность более точно оценить метрику расширения, так как самый короткий маршрут между узлами вычислительной сети может оказаться не самым вероятным.

Метрика перегруженности позволяет оценить количество пар взаимодействующих потоков выполнения, использующих ребро, связывающее две точки из ***VH***. При расчёте этой метрики требуется использование функции маршрутизации *RH*, а если она не зада- на, то вероятность отправки пакета по кратчайшему маршруту считается равной 1, по прочим маршрутам – 0. Пусть *pl* (*u*,*v*) – вероятность прохождения какого-либо из маршрутов между *u* и *v* через ребро 𝑙 ∈ 𝑉𝐻, где 𝑢, 𝑣 ∈ 𝑉𝐴. Тогда мы можем определить метрику перегруженности для ребра *l* как

Стоит отметить, что в случае, когда потоки *ui*, *vj* не взаимодействуют между собой, *pl* (*ui*,*vj*) = 0 для всех *l*.

Затем задаётся функция, позволяющая оценить метрику перегруженности для всей сети. Одна из распространённых метрик такого рода – максимальная перегруженность:

Рассмотренная задача минимизации относится к классу *квадратичных задач о назначениях* и является NP-трудной, поэтому оптимальное решение в общем случае может быть найдено лишь при очень малом размере ***VA*** и ***VH***. Следует отметить, что оптимальное решение этой задачи может быть найдено за полиномиальное время в частных случаях, например, когда виртуальная топология и топология вычислительное сети имеют форму куба или форму отрезка прямой. При решении задачи сопоставления топологий в общем случае обычно ограничиваются нахождением субоптимального решения. Рассмотрим некоторые из используемых для этого алгоритмов:

1. Жадные алгоритмы. На первом шаге выбираются начальные точки 𝑢0 ∈ 𝑉𝐻, 𝑣0 ∈ 𝑉𝐴, создаётся отображение σ: *v*0 → *u*0. На *i*-м шаге выбирается по одной ближайшей к *ui–*1 и *vi–*1 точке, создаётся отображение между ними σ: *vi* → *ui*
2. Разбиение графа. Выполняется рекурсивное разбиение графа на *k* частей. Затем запускается обратный ход рекурсии, сопровождающийся формированием отображения σ.
3. Увеличение степени близости графов. Алгоритмы этого типа оперируют матрицами смежности, построенными для ***VA*** и ***VH***, уменьшая их *пропускную способность*1. Таким образом, обе матрицы смежности приводятся к каноническому виду, что делает рассматриваемые графы близкими в определённом смысле и позволяет провести для них сопоставление топологий.
4. Поиск изоморфного подграфа. Предполагается, что **V*H*** содержит больше узлов, чем **V*A***, поэтому задача сопоставления топологий сводится к поиску подграфа **V*H***, изоморфного графу **V*A***.

После того, как проведено сопоставление топологий, требуется распределить потоки выполнения по ядрам процессора каждого вычислительного узла. При этом важно выполнить привязку потоков к ядрам, чтобы исключить возможность переноса потока на другое ядро процессора. В случае такого переноса может значительно увеличиться количество промахов кэша.

**Динамическая виртуальная топология**. Представим теперь ситуацию, когда перед началом исполнения распределённого приложения его потоки выполнения распределены по узлам вычислительной сети в соответствии с результатами решения рассмотренной задачи минимизации, но в ходе работы приложения потребовалось динамически создать поток выполнения, отсутствовавший в исходной виртуальной топологии. В этом случае либо при выходе из строя одного из узлов вычислительной сети можно заново решить задачу минимизации для изменённого множества потоков выполнения ***A*'** (либо изменённого множества узлов ***H*'**), но в результате может понадобиться перераспределить по узлам вычислительной сети уже запущенные потоки выполнения. Чтобы этого избежать, на практике вместо нового решения задачи минимизации используют некую эвристику, чтобы поместить новый поток выполнения «близко» к потокам, с которыми он будет взаимодействовать.

### **1.2.4 MapReduce**

Рассмотрим одну из распространенных технологий облачных вычислений MapReduce. MapReduce предназначена для создания и обработки больших объёмов данных с помощью параллельного алгоритма на кластере. Обрабатываемые данные могут храниться в файловой системе или БД. Для изложения сути этой технологии необходимо ввести некоторые определения.

*Функция высшего порядка* – в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата.

Технология MapReduce основана на использовании двух функций высшего порядка – *map*() и *reduce*().

*Map* – функция высшего порядка, которая применяет переданную в качестве аргумента функцию к каждому элементу списка, переданного в качестве другого аргумента. *Map* возвращает список, элементом которого является результат выполнения функции- аргумента.

*Reduce* (*свёртка*) – функция высшего порядка, которая производит преобразование структуры данных к единственному атомарному значения при помощи заданной функции.

Технология MapReduce предполагает наличие *центрального* (*главного*) узла, который координирует выполнение задач *рабочими* узлами. Обработка данных в MapReduce проводится по следующему обобщённому алгоритму.

Шаг 1. Подготовка входных данных для функции *map*. Центральный узел назначает узлы, которые будут выполнять операцию map, и передаёт им входные данные. Каждый узел получает данные, соответствующие ключу *Ki*.

Шаг 2. Выполнение пользовательской функции, переданной в функцию *map*.Функция *map* выполняется рабочими узлами для каждого ключа *Ki*: *Ti* = *map*(*Ki*).

Шаг 3. Распределение данных по reduce-узлам.Центральныйузел

группирует данные, полученные от map-узлов **T = group(T),** назначает узлы, которые будут выполнять операцию reduce, и передает им данные T.

Шаг 4. Выполнение пользовательской функции, переданной в функцию *reduce*. Функция *reduce* выполняется рабочими узлами для каждого значения *Tʹi: Ri* = *reduce*(*Tʹi*).

Шаг 5. Вычисление конечного результата. Центральный узел собирает все *Ri*, сортирует их и вычисляет финальное значение.

Можно логически представлять себе эти шаги, как выполняемые последовательно, но на практике они могут чередоваться, если количество входных данных слишком велико для их эффективной обработки на заданном наборе узлов за один проход алгоритма.

Существуют несколько реализаций MapReduce, в частности:

* Google MapReduce, реализованный на C++, с интерфейсами на Python и Java.
* Apache Hadoop MapReduce, реализованный на языке Java.

Канонический пример программы, написанной с помощью MapReduce – программа, подсчитывающая частоту, с которой встречается каждое слово в наборе документов.

Функция *map*() здесь генерирует набор пар (ключ, значение), где ключом является слово, а значением цифра 1. Функция *reduce*() подсчитывает суммарную частоту, с которой встречается заданное слово.

Вначале центральный узел распределяет документы по рабочим узлам. На 2-м шаге они выполняют функцию map для полученных данных. На 3-м шаге центральный узел группирует результаты 2-го шага по словам, для которых проводится подсчёт частоты встречаемости, и распределяет сгруппированные данные по рабочим узлам.

При этом происходит балансировка нагрузки на рабочие узлы. Так на 4-м шаге функцию *reduce*() для часто встречающегося слова может выполнять сразу несколько узлов, и один узел может обрабатывать данные для нескольких редко встречающихся слов. В конце центральный узел суммирует частоты (функция *finalStage*()), полученные на различных узлах для каждого слова, формируя результирующую структуру данных.

## **1.3 Распределенные файловые системы**

Распределенная файловая система (РФС) – это клиент серверное приложение, которое позволяет клиенту хранить и обращаться к данным, сохранённым на сервере так, как если бы эти данные хранились локально на клиентской стороне. Когда пользователь обращается к области файла, сервер посылает клиенту её копию, которая кэшируется на клиентской стороне. Если пользователь изменил файл, то по окончании работы с ним изменения пересылаются обратно на сервер. Серверная сторона современных РФС представлена не единственным сервером, а сетью таких серверов. РФС, как правило, используют механизм *репликации данных* для повышения отказоустойчивости. Если файл доступен нескольким клиентам, то РФС должна реализовывать механизм синхронизации доступа к файлам так, чтобы клиенты всегда работали с последней версией файла, и не возникал конфликт изменений, который возможен при попытке модификации файла одновременно несколькими клиентами.

РФС отличается от *распределённого хранилища данных* тем, что для доступа к распределённым данным первая использует тот же интерфейс, что и для доступа к локальным данным. Такой интерфейс включает получение списка файлов и папок в текущей папке, побайтовые операции чтения и записи, использование системных прав доступа. Распределённое хранилище данных, напротив, требует использования отдельного программного интерфейса (англ. Application Programming Interface, API) и имеет семантику, отличную от файловой системы, например, семантику базы данных. Рассмотрим некоторые известные реализации РФС.

### **1.3.1 Google File System**

Google File System (GFS или GoogleFS) разработана корпорацией Google для внутреннего использования. В частности, GFS применяется для обеспечения функционирования поискового механизма, который может генерировать огромный объём данных.

Поэтому GFS оптимизирована для случая частого добавления новых файлов и добавления данных в конец существующих файлов. В то же время GFS хуже приспособлена для выполнения операции удаления файлов или части данных из файлов. GFS также рассчитана на функционирование на вычислительных кластерах Google, в которых широко применяются ПК в качестве узлов, что приводит к частому выходу отдельных узлов из строя и сопутствующим потерям данных.

Стоит отметить, что в GFS акцент сделан на высокой пропускной способности, достигаемой в том числе за счёт снижения времени отклика.

Для понимания возможностей масштабирования GFS достаточно сказать, что в самый большой кластер под управлением этой файловой системы содержал сотни терабайтов данных на тысячах узлов, обеспечивая одновременное обслуживание сотен клиентов.

Узлы в GFS бывают двух типов – *ведущий сервер* (англ. *Master*) и *сервер данных* (англ. *Chunkserver*). Файлы разделяются на блоки фиксированного размера (англ. *chunk*) по 64 мегабайта.

Такие блоки хранятся на серверах данных. Ведущий сервер сопоставляет каждому блоку 64-битный идентификатор в момент создания.

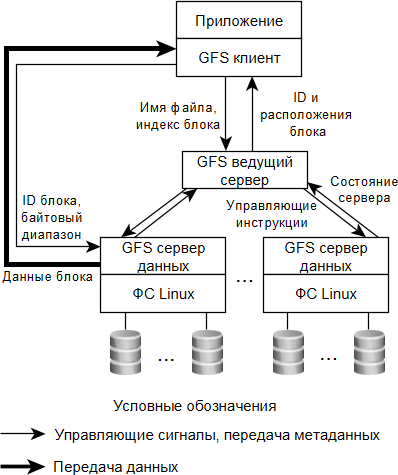
Для каждого блока создаётся несколько копий (три по умолчанию), которые хранятся на различных серверах данных. Клиентское приложение может задать требуемое количество копий файла при создании, а также изменить значение этого параметра в дальнейшем.

Рисунок 1.3.1 - Архитектура GFS

При выборе серверов данных для **размещения нового блока** ведущий сервер руководствуется следующими правилами для балансировки нагрузки на серверы и повышения надёжности системы.

1. Выбираются серверы с уровнем заполнения жёсткого диска ниже среднего.
2. Ведущий узел старается минимизировать количество операций создания нового блока для каждого сервера данных за единицу времени. В то время как сама операция создания блока не является трудоёмкой, за ней следует трудоёмкая операция записи.
3. Серверы данных должны быть расположены в различных сегментах сети, чтобы минимизировать вероятность одновременной потери связи с ведущим узлом сразу несколькими из них.

Ведущий сервер периодически выполняет восстановление балансировки существующих блоков по таким параметрам, как заполнение диска на серверах данных и нагрузке между ними. Через этот механизм происходит постепенное заполнение нового подключенного сервера данных. Заполнение такого сервера не становится лавинообразным за счёт применения ведущим сервером тех же правил, что и при создании нового блока, а именно 2-го правила. Восстановление балансировки также может потребоваться при сравнительно длительной высокой нагрузке на определённый сервер данных или сегмент сети.

Одна из копий данных является *основной*, другие – *резервными*. Ведущий сервер не хранит блоки, а содержит метаданные, включающие таблицы соответствия блоков и файлов, расположение копий блоков. К таким метаданным также относится список процессов, которые в данный момент выполняют запись и чтение блоков или копирующих блок на ещё один сервер данных, что происходит при выходе из строя одного из серверов данных. Стоит отметить, что ведущий сервер хранит все метаданные в ОП, что повышает возможности оперативной обработки данных. При этом ведётся журнал событий, который содержит историю изменения метаданных и хранится на жёстком диске. Периодически создаются контрольные точки, содержащие состояние метаданных с учётом всех предыдущих изменений. В момент создания контрольной точки происходит переключение на новый журнал событий.

**Ведущий сервер**. При инициализации ведущего сервера текущее состояние восстанавливается по последней контрольной точке и журналу, содержащему последующие события. Использование контрольных точек позволяет минимизировать время инициализации. Чтобы не блокировать ведущий сервер на время создания контрольной точки, соответствующие операции выполняются в отдельном потоке, при этом параллельно идёт формирование нового журнала событий.

Ведущий узел отвечает на запрос, требующий изменения состояния, только после того как внёс запись в журнал состояний как локально, так и в нескольких *теневых ведущих* узлах. Как правило, одновременно записывается серия изменений, чтобы уменьшить количество дисковых и сетевых операций.

Ведущий сервер поддерживает метаданные в актуальном состоянии, периодически получая обновлённые данные от каждого сервера данных с помощью сообщения HeartBeat. Это сообщение позволяет ведущему серверу не хранить данные о расположении блоков данных на своём диске, а запрашивать их при инициализации.

При выходе из строя ведущего сервера мониторинговая утилита, не входящая в состав GFS, делает активным один из теневых ведущих серверов. Клиенты используют DNS-псевдоним ведущего сервера, поэтому их не приходится оповещать об изменении его расположения. Для повышения скорости переключения ведущего сервера, теневые сервера поддерживаются в актуальном состоянии. Так, теневые сервера не только сохраняют историю изменения метаданных, но и применяют получаемые изменения, а также выполняют опрос серверов данных при старте и затем периодически, хотя и реже, чем ведущий сервер. Стоит отметить, что теневые сервера предоставляют доступ на чтение, в том числе в то время, когда ведущий сервер недоступен. Как правило, изменения в теневых серверах запаздывают на доли секунды, поэтому клиенты обращаются к ним для получения данных о расположении блоков, которые не претерпевают изменений в данный момент, либо в случае, когда приложение готово получать несколько устаревшие данные. В качестве примера можно привести приложение, которое считывает по мере поступления блоки файла, постепенно дополняемого другими приложениями.

**Операции над данными**. Рассмотрим теперь выполнение операций чтения и записи более подробно. Диапазон адресов в файле транслируется в номера блоков, используя информацию о фиксированном размере блока. Эти номера блоков включаются в запрос к ведущему серверу. Ведущий сервер в ответ высылает информацию о расположении узлов данных, содержащих запрошенные блоки файла, а также следующие за запрошенными. Это позволяет сократить количество запросов от клиентов к ведущему серверу, так как приложения, использующие GFS, в большинстве случаев последовательно считывают большие области файлов. Полученная информация кэшируется на определённый интервал времени, по истечении которого необходимо выполнить новый запрос к ведущему узлу. При ***чтении*** данных клиент, как правило, для каждого блока выбирает ближайший сервер данных для скачивания.

При ***записи*** блока данных сначала происходит его конвейерная отправка на все сервера данных *C*, на которых он должен быть сохранён. При этом каждый сервер пересылает данные следующему ближайшему серверу из множества *C*, начиная отправку уже при получении первого пакета данных, не дожидаясь получения всего блока. Затем клиент посылает запрос на запись отосланного блока серверу, хранящему основную копию блока. Этот узел пересылает запрос на запись всем серверам, хранящим резервную копию блока, и оповещает клиента об успешном завершении записи только после получения подтверждения об этом от всех остальных серверов. В противном случае осуществляется ещё одна попытка записи блока.

Операции записи в пределах одного блока данных выполняются в порядке их поступления на ведущий сервер. Но поскольку запись данных проводится по указанному клиентом отступу от начала файла, то при параллельной записи данных в файл несколькими клиентами легко может возникнуть ситуация, когда изменения, сделанные одним клиентом, будут затёрты изменениями от другого клиента. Поэтому GFS поддерживает операцию добавления данных (англ. ***append***), которая является атомарной и гарантирует добавление переданной порции данных целиком как минимум один раз в конец файла без перекрытия с другими порциями данных, записываемых параллельно. Операция добавления данных возвращает клиенту позицию переданной им порции данных в файле. Размер такой порции ограничен 1/40 от размера блока, который, как мы помним, составляет 64 МБ, и её запись всегда производится в пределах одного блока.

Если произошла ошибка при добавлении данных хотя бы в одну из резервных копий файла, клиенту возвращается ошибка, и операция повторяется. В этом случае в некоторых копиях файла может оказаться несколько копий записываемой порции данных. В то же время, поскольку при успешном выполнении операции добавления данных во всех резервных копиях новая порция данных должна иметь одну и ту же позицию в файле, в некоторых резервных копиях могут возникнуть участки, не содержащие значимых данных. В результате GFS не гарантирует побайтового равенства резервных копий, и клиентские приложения должны быть к этому готовы. Так, добавляемые порции данных должны содержать средства их идентификации и проверки наличия повреждений, что может выражаться в сопоставлении каждой из них уникального идентификатора и контрольной суммы.

GFS поддерживает операцию получения моментального снимка (англ. ***snapshot***) файла или директории. Эта операция выполняется отложенным способом, то есть блоки файла копируются только при попытке их изменения. Так при запросе клиента к ведущему серверу на запись, сначала он посылает всем серверам данных, хранящим копии соответствующего блока, сообщение, чтобы они создали его копию, и лишь после этого высылает клиенту информацию о расположении серверов данных.

При получении запроса на ***удаление*** файла, ведущий сервер вносит изменения в метаданные, переименовывает файл и делает его скрытым, но не удаляет файл физически. Такой файл хранится в течение трёх дней с момента получения запроса на удаление (интервал времени является настраиваемым). В течение этого интервала файл можно восстановить, после чего информация о нём удаляется из метаданных ведущего сервера, что приводит к появлению *висячих блоков* (не связанных ни с каким файлом). Ведущий сервер на регулярной основе проверяет, какие блоки стали висячими, и удаляет информацию о них из метаданных. Затем, обработав ответ на запрос HeartBeat, содержащий список блоков на сервере данных, ведущий сервер отправляет ему список блоков, которые могут быть удалены.

Такой способ удаления файлов, называемый ещё *сборкой мусора*, позволяет не только восстанавливать файлы при их ошибочном удалении, но и балансировать нагрузку как на ведущий сервер, так и на сервера данных за счёт выполнения удаления блоков в период наименьшей нагрузки на систему. В то же время такой способ реализации удаления файлов может быть неэффективным, когда приложение создаёт и удаляет множество временных файлов. Для этого случая предусмотрена настройка, позволяющая не создавать резервные копии отдельного файла или файлов внутри директории, а также удалять файл мгновенно без применения описанной выше процедуры.

Для поддержания целостности данных на сервере данных блок разбивается на *мини блоки* по 64 Кб, для каждого из которых вычисляется 32-битная контрольная сумма. При чтении данных это значение сверяется с вновь вычисленным для всех затронутых мини блоков, а при записи – для первого и последнего мини блоков. Сверка контрольной суммы при записи позволяет исключить возможность скрытия факта искажения данных при пересчёте контрольной суммы. При несовпадении контрольной суммы сервер данных возвращает код ошибки клиенту, чтобы он обратился к резервной копии, и оповещает об этом ведущий сервер, который заменяет повреждённый блок резервной копией. В период простоя серверы данных вычисляют и проверяют контрольную сумму для блоков, не использовавшихся в последнее время. При обнаружении ошибки оповещается ведущий сервер.

Блоки файлов не кэшируются на стороне клиента, потому что типовые варианты использования предполагают либо последовательное чтение больших файлов, либо работу с большим набором данных, который бы не поместился в кэше. Такой подход упрощает реализацию клиента, так как отпадает необходимость в поддержании когерентности кэша. Серверы данных также не имеют кэша на уровне GFS, так как блоки хранятся в виде файлов в ОС Linux, файловая система которой реализует кэширование в ОП часто используемых файлов.

Коммуникационные протоколы GFS базируются на стеке протоколов TCP/IP. Клиентская часть GFS реализована в виде программной библиотеки.

Переосмысление области применения. В процессе использования GFS выявилось несколько проблем. Первая из них заключалась в том, что некоторые приложения создают значительное количество файлов небольшого размера, что не предусмотрено архитектурой GFS. Так как все метаданные должны храниться в ОП ведущего сервера (единственного в кластере), создание большого количества файлов малого размера приводит к исчерпанию его ОП задолго до полного заполнения кластеров данных. Второй проблемой для GFS стало появление интерактивных приложений, таких как Gmail, которым требуется не только высокая пропускная способность, но и быстрый отклик РФС.

Первая проблема была во многом решена путём создания распределённой СУБД BigTable, построенной на основе GFS. Теперь приложения получили возможность сохранять структурированные данные в BigTable, создавая столбцы таблицы вместо разрозненных файлов, что привело к существенному сокращению количества файлов, хранимых в GFS. Вторая проблема решается разработчиками приложений, использующих GFS, таких как Gmail, путём хранения данных одновременно в нескольких кластерах. Это даёт возможность перенаправлять запросы другому кластеру, если временно возрастает время отклика текущего кластера.

**1.4 Универсальные вычисления на видеокарте**

**Архитектура современных видеокарт.** Производительность современных дискретных видеокарт которыми оснащена значительная часть ПК, измеряется терафлопами, в то время как производительность центрального процессора – десятками гигафлопов. ЦП не участвует в выполнении программ на видеокарте, поэтому его ресурсы высвобождаются для выполнения других задач, что открывает возможности для организации гетерогенных вычислений. Направление информатики, посвящённое способам использования вычислительных ресурсов видеокарты для решения задач, не связанных напрямую с визуализацией, называется *универсальные вычисления на видеокарте* (англ. General-Purpose computation on Graphics Processing Units*,* GPGPU). GPGPU позволяет перенести большой объём вычислений с ЦП на видеокарту.

Архитектура видеокарты такова, что подавляющая часть транзисторов сосредоточена в блоке арифметико-логических устройств (англ. arithmetic logic unit), и лишь незначительная часть транзисторов обеспечивает кэширование и выполнение инструкций потока управления. Таким образом, архитектура видеокарты рассчитана на интенсивные арифметические и логические вычисления, но плохо приспособлена для алгоритмических задач с большим количеством условных переходов.

Вот неполный список сфер, в которых находят своё применение вычисления на видеокарте: вычислительная гидрогазодинамика, автоматизация проектирования электроники, финансы, физика для игр, графика, вывод изображений, визуализация в медицине, численные методы, обработка сигналов, обработка видео - и аудио - данных.

* + 1. **Обзор программных интерфейсов**

Рассмотрим технологии, применяемые в GPGPU. Исторически первой технологией, позволяющей отойти от модели жёстко зафиксированного *графического конвейера*, стали *шейдеры* – программы, предназначенные для исполнения процессорами видеокарты. *Пиксельные шейдеры* больше других (вершинных и геометрических) подходят для осуществления общих вычислений на видеокарте, так как позволяют получить программный доступ к отдельным пикселям изображения. Но пиксельные шейдеры проектировались для визуализации изображений, что значительно сужает их область применения для общих вычислений.

Подходы к ускорению программы с помощью вычислительных ресурсов видеокарты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Подход к уско- рению програм- мы | Реализации | Сложность встраивания в существующую программу | Гибкость использова- ния |
| 1 | Библиотеки, обрабатываю- щие данные на  видеокарте | cuBLAS, NVBLAS,  nvJPEG, cuFFT, nvGRAPH, cuRAND и др. | Низкая | Низкая |
| 2 | Директивы компилятора | Microsoft C++ Accelerated Massive Parallelism  (C++ AMP), OpenACC  (Open Accelerators) | Средняя | Средняя |
| 3 | Языки програм- мирования | Nvidia CUDA (C++, Python, Fortran), Open Computing Language  (OpenCL), DirectCompute | Высокая | Высокая |

Таблица 1.4.1

Затем были разработаны более гибкие технологии, позволяющие перенести на видеокарту обширный спектр задач. Рассмотрим основные программные интерфейсы для доступа к вычислительным ресурсам видеокарты, перечисленные в табл.

Подходы к ускорению программы в таблице ранжированы по сложности встраивания в существующую программу. Первый подход предполагает встраивание в программу библиотеки, осуществляющей доступ к вычислительным ресурсам видеокарты. Этот подход является наименее трудоёмким для программиста, но при этом характеризуется наименьшей гибкостью, так как возможности применения общих вычислений на видеокарте ограничены интерфейсом подключённой библиотеки.

Второй подход, предполагающий использование директив компилятора, характеризуется сбалансированными показателями сложности встраивания в существующую программу и гибкости использования. Стандарт OpenACC описывает набор директив компилятора, предназначенных для создания гетерогенных параллельных программ, задействующих как центральный, так и графический процессор. Эти директивы используются для аннотирования фрагментов кода на C, C++ и Fortran. С помощью OpenACC-директив программист определяет, какие участки кода должны выполняться на ускорителе, в какой момент должно происходить копирование данных из ОП в видеопамять и обратно. В качестве такого ускорителя могут выступать видеокарты Nvidia, AMD Radeon, а также многоядерный процессор Intel Xeon Phi. К плюсам этого подхода относится возможность итеративного распараллеливания существующей программы, а также отсутствие необходимости в дублировании кода, так как одни и те же процедуры могут выполняться как на ЦП, так и на видеокарте.

Третий подход, предполагающий применение специализированных языков программирования, позволяет программисту проводить тонкую настройку выполнения, а также оптимизацию программы на видеокарте, но требует максимального времени на адаптацию существующего кода к параллельному исполнению. Nvidia CUDA (англ. Compute Unified Device Architecture) и OpenCL (англ. Open Computing Language) предполагают написание программ на расширенных версиях языка C++, в то время как Di- rectCompute представляет собой API, доступное из различных языков программирования. Как правило, использование таких сравнительно низкоуровневых средств доступа к вычислительным ресурсам видеокарты позволяет программисту достичь лучшего понимания архитектурных особенностей графического ускорителя, а также быстрее научиться оценивать возможности аппаратного ускорения тех или иных частей программы.

Таким образом, облачные технологии является перспективной сферой в предоставление вычислительних мощностей, но на данный момент, чтобы получить достойный уровень вычислений, нужно потрать много денег. Да и такие мощности нужны далеко не всем.