# **Tема 1. Введение в распределенные ИС**

## **1. Понятие системы. Понятие информационной системы. Централизованные и распределенные системы. Понятие распределенной системы.**

**Система:** множество связанных элементов. Характеризуется общей целью (назначением), набором задач (функций) для достижения цели.

**ИС:** социально-техническая система для сбора, хранения, поиска, передачи и обработки информации. Есть цель. Пример: информационная система продажи билетов; цель – увеличить коммерческую загрузку рейсов.

Централизованные - ИС из 1 компьютера и переферии => процесс обработки инф централизованный.

**РИС** - система, взаимодействующие элементы (узлы) которой пространственно (географически) отделены друг от друга, но пользователю представляется как единое целое.

## **2. Основные характеристики распределенных ИС: - Коллекция автономных вычислительных элементов, оверлейные сети; Единая согласованная система.**

**Хар-ки РИС**

1. «Все узла могут действовать независимо друг от друга»
2. Результатом независимости узлов являются:

Независимость времени

Возможность группового управления узлами

***- Коллекция автономных вычислительных элементов, оверлейные сети;***

В результате организации коллекций на практике часто возникают т.н. оверлейные сети (покрывающие сети). При этом могут встречаться 2 вида перекрытия:

**Структурное** – в этом случае узлы хорошо осведомлены о своих соседях с которыми они могут поддерживать коммуникации.

**Не структурное (без структурное**) – в этом случае каждый узел имеет определенное число ссылок на соседей, но выбирает их случайным образом.

Оверлейные сети всегда связные, т.е. между двумя любыми узлами всегда имеется путь.

Примером оверлейных сетей являются сети Р2Р (peer-to-peer)

***Единая согласованная система.***

Представление РаспределеннойСиситемы как единой системы порождает у пользователя ощущение «прозрачности» системы.

С другой стороны отказ одного из узлов РС может привести к отказу всей системы в целом, что порождает проблему надежности и восстановления после отказов.

## **3. История возникновения и развития распределенных ИС. Эволюция РИС.**

1960-е – Появление ОС, реализующих межпроцессное взаимодействие

1960-е – ARPANET и далее Usenet, FidoNet, Internet…

1960-1970-е – появление многопроцессорных систем

1970-е – Широкое распространение ЛВС на базе Ethernet

1980-е – Распределенные вычисления становятся активно изучаемой и обсуждаемой темой. Начало использования модели взаимодействия клиент-сервер в корпоративных РИС.

1990-е – Интернет становиться доступным для коммерческих организаций и частных лиц. Появление Объектной Архитектуры и Архитектуры распределенных объектных архитектур. Корпоративные ИС повсеместно становятся распределенными.

1994 – Трафик WWW превысил объем трафика электронной почты.

1994-1995 – Появление 3-х звенной архитектуры распределенных Web-приложений

1995 – Начало использования Интернет технологий в корпоративных РИС. Возникновение понятия intranet.

1996 – Появление вычислительных кластеров на базе IBM PC (проект Beowolf)

Конец 1990-х появление peer-to-peer систем Интернет

1998 – Создание языка XML.

1998-2000 – Разработка технологий виртуализации для архитектуры Intel\_x86

1999 – 2005 Появление и становление GRID систем. Появление Web-сервисов на основе XML/SOAP

2000-2002 Появление RESTFull архитектуры WEB-сервисов.

2002-2004 Появление сервис ориентированной архитектуры (СОА). Разработка стандартов на веб-сервисы.

2000-2005 Появление мобильных устройств и ОС для них.

2005 – Появление облачных вычислений.

2006-2010 – Возникновение публичных облачных сервисов AWS, Google, Azure и других.

2010 – 2013 – Появление микросервисной архитектуры.

2010 – 2015 – появление технологий непрерывного развертывания на основе DevOps.

2015-2020 – широкое распространение облачных технологий и технологий Internet of Things (Интернет вещей)

## **4. Классификация РИС.**



**По архитектуре**

–Клиент-серверная (client-server)

–3-tier / N-tier

–Одноранговая (peer-to-peer)

–Кластерная (сильно связанная)

–С общим адресным пространством

**По территории**

–Локальная

–Городская

–Глобальная

**По степени автоматизации**

–Автоматизированные

–Автоматические

**По способу обработки данных**

–Информационно-справочные

–Вычислительные (обработки данных)

**По наличию центрального компонента**

–С центральным компонентом (клиент/сервер)

–Без центрального компонента (perr-to-peer)

**По реактивности**

–Реального времени

–Без жестких требований ко времени реакции

## **5. Преимущества и недостатки РИС. Особенности современных РИС.**

**+**

Получения доступа к физически недоступным ресурсам

Увеличение производительности системы по сравнению с единым локальным приложением

Синергетические выгоды – самоорганизация в РС

Высокая надежность, за сет дублирования всех важнейших компонентов.

**-**

Увеличенное время реакции системы

Сложность контроля удаленных элементов

Сложность разработки, отладки и использования

Дополнительные усилия по обеспечению информационной безопасности

Низкая надежность

## **6. Состав РИС. Виды обеспечения РИС. Корпоративные РИС.**

***Состав:*** Хосты / компьютеры, Сеть / средства коммуникации, Приложения, Данные, Пользователи

***Виды:***

Информационное

Техническое

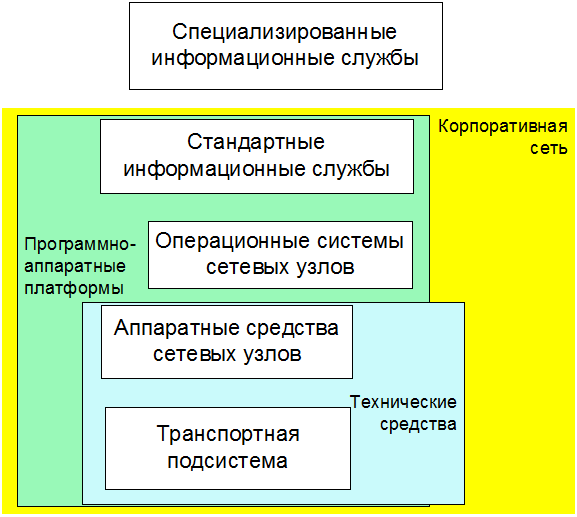
Математическое и программное

–ОПО

–СПО

Организационное

Правовое



## **7. Основные задачи создания и свойства РИС: - Доступность; прозрачность; открытость; гибкость; масштабируемость РИС.**

**Цели создания рис:**

* Поддержка распределенного доступа к ресурсам
* Обеспечение следующих свойств РИС
* Обеспечение прозрачности
* Обеспечение открытости
* Обеспечение масштабируемости
* Обеспечение безотказности работы

**Задачи рис:**

* Обеспечить связь пользователей с ресурсами РИС
* Ресурсы распределенных информационных систем: вычислительные ресурсы (процессорное время), информационные ресурсы (файлы, базы данных), принтеры и пр.
* Обеспечить прозрачность РИС. Системы, которые представляются пользователям и приложениям в виде единой системы, называются прозрачными.
* Обеспечить открытость РИС. Системы построенные на открытых стандартах называются открытыми
* Обеспечить масштабируемость РИС. Системы допускающие рост ее размеров без потери свой производительности называется масштабируемой.

## **8.** **Технологии масштабирования: вертикальное и горизонтальное масштабирование; сокрытие времени ожидания; распределение; репликации и кэширование.** **Масштабируемость** – сохранение работоспособности системы при изменении масштаба системы.

Масштабируемость системы может измеряться по трем показателям:

* По размеру.
* По географической распределенности.
* По степени сохранения управляемости при увеличении масштаба РИС

**Вертикальное масштабирование** – увеличение производительности каждого компонента системы с целью повышения общей производительности.

–возможность заменять в существующей вычислительной системе компоненты более мощными.

**Горизонтальное масштабирование** – возможность добавлять к системе новые узлы (серверы, процессоры) для увеличения общей производительности.

**Имеется три основные технологии масштабирования:**

* Сокрытие времени ожидания выполнения удаленного запроса (времени ожидания связи);
* Распределение компонентов или их частей по различным узлам системы (организация параллельной работы компонент)
* Репликация данных и компонент системы. Кэширование – особая форма репликации.

## **9. Ловушки при разработке распределенных ИС.**

В отличие от традиционного ПО, компоненты РС распределены по сети. В результате возникает большое число причин возможных отказов. Петер Дойтч во время его работы в Sun сформулировал основные ошибочные допущения разработчиков:

* Надежность сети;
* Безопасность сети;
* Гомогенность сети;
* Неизменность топологии сети;
* Отсутствие задержек при передаче по сети;
* Неизменность пропускной способности сети;
* Нулевая стоимость транспорта по сети;
* Наличие единого администратора сети.

# **Тема 2. Введение в распределенные ИС.**

## **10. Типы распределенных систем.**

Типы современных РС сформировались в процессе эволюции двух классов систем:

* Систем высокопроизводительных вычислений (HPC)
* Систем высокой пропускной способности (HTC)

Типы распределенных систем

* Системы высокопроизводительных распределенных вычислений (High performance distributed computing).
* Распределенные информационные системы (Distributed information systems).
* Повсеместные - всеобъемлющие системы (Pervasive systems). Это мобильные и встраиваемые системы, сюда же относят системы слежения, обладающие сетью датчиков и множеством исполнительных механизмов и др. РС.

## **11. Концепции аппаратных решений для РИС: - набор независимых машин; многопроцессорные системы; вычислительный кластер (многомашинные. Классификация ВС Флинна.**

Все компьютеры можно разделить на две группы. Системы, в которых

компьютеры используют память совместно, обычно называются **мультипроцессорами**, а работающие каждый со своей памятью — **мультикомпьютерами**.

Это вычислительные системы, обмен данными между которыми реализуется через устройства ввода вывода. Сегодня такими устройствами являются сетевые карты.

Современные **многомашинные системы** представляют собой множество дешевых серверов, объединяемых компьютерной сетью.

**Классификация ВС Флинна:**

Среди всех рассматриваемых систем классификации ВС наибольшее признание получила классификация, предложенная в 1966 году.

В ее основу положено понятие потока, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором.

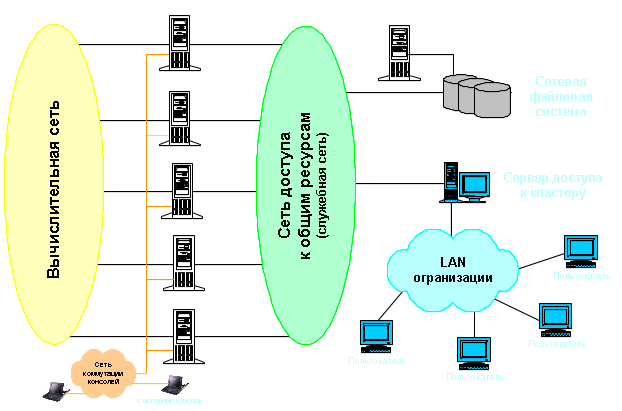
В зависимости от количества потоков команд и потоков данных Флинн выделяет четыре класса архитектур:

* **SISD** (Single Instruction stream/Single Data stream) - cистемы с одиночным потоком команд и одиночным потоком данных (ОКОД);
* **MISD** (Multiple Instruction stream/Single Data stream) - cистемы с множественным потоком команд и одиночным потоком данных (МКОД);
* **SIMD** (Single Instruction stream/Multiple Data stream) - cистемы с одиночным потоком команд и множественным потоком данных (ОКМД);
* **MIMD** (Multiple Instruction stream/Multiple Data stream) - cистемы с множественным потоком команд и множественным потоком данных (МКМД)

## **12. Кластерные системы Beowulf. Архитектура кластерной системы.**

Одним из примеров кластерных систем являются кластера Beowulf.

В них различают управляющий и вычислительные узлы, объединяемые в многомашинную систему с помощью компьютерной сети.



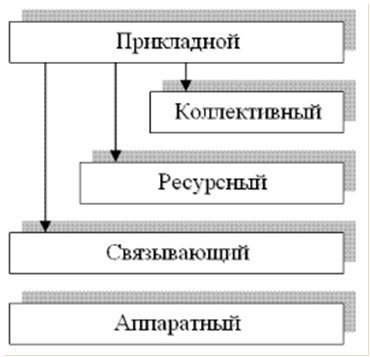
## **13. Grid системы. Архитектура GRID. Облачные вычисления**.

**Грид –** это система, которая координирует распределенные ресурсы посредством стандартных, открытых, универсальных протоколов и интерфейсов для обеспечения нетривиального качества обслуживания (QoS – Quality of Service).

Концепция грид-вычислений идея подобна концепции электросети: нам не важно, откуда к нам в розетку приходит электричество. Независимо от этого мы можем подключить к электросети утюг, компьютер или стиральную машину.

Аналогично и в идеологии грид: мы можем запустить любую задачу с любого компьютера или мобильного устройств на вычисление, ресурсы же для этого вычисления должны быть автоматически предоставлены на удаленных высокопроизводительных серверах, независимо от типа нашей задачи.

Архитектура Grid систем



* Ключевой проблемой grid является объединение ресурсов из различных организаций для совместного использования. В результате формируется федерация систем, которая влечет за собой образование виртуальных организаций.
* Для построения Grid была разработана и принята в качестве стандарта *OGSA (Open* Grid Services Architecture – Открытая архитектура грид-сервисов), основанная на архитектуре SOA – Service Oriented Architecture.

**Облачные вычисления.**

Является развитием концепции Utility Computing – предоставление обслуживания запросов по заявкам пользователей с оплатой, основанной на учете использованных ресурсов.

Облачные вычисления характеризуются предоставлением пользователю легко доступных виртуализированных ресурсов.

Пользователь, подключаясь к облачному ресурсу, сам определяет, что он хочет получить, определяет ресурсы, а затем получает, после подтверждения/регистрации.

## **14. Всеобъемлющие (всепроникающие) системы (Pervasie systems) и Вездесущие (Ubiquitous systems) системы. Краткая характеристика и особенности этих систем**

**Всеобъемлющие системы.** К этому типу РИС относятся:

* Вездесущие (Ubiquitous systems) системы – пользователи даже не подозревают, что взаимодействуют с ними. Например, системы организации движения в городах, системы видеонаблюдения и т.п.
* Мобильные компьютерные системы.
* Системы сенсоров (датчиков).
* Интернет вещей (Internet of Things – IoT).

**Вездесущие системы.** Основные особенности этих систем:

* Распределенность в пространстве, узлы системы объединены сетью, между ними обеспечивается прозрачный доступ.
* Интерактивный обмен между пользователями и устройствами системы – носит ненавязчивый характер.
* Автономность. Устройства действуют автономно без взаимодействия с людьми.
* Интеллектуальность. Система в целом может управлять широким составом действий и взаимодействий.
* Пример: системы управления автомобилями и др.

## **15. Программные системные решения для РИС: - Распределенные ОС; Сетевые ОС; ПО промежуточного слоя (уровня). Краткая характеристика этих решений.**

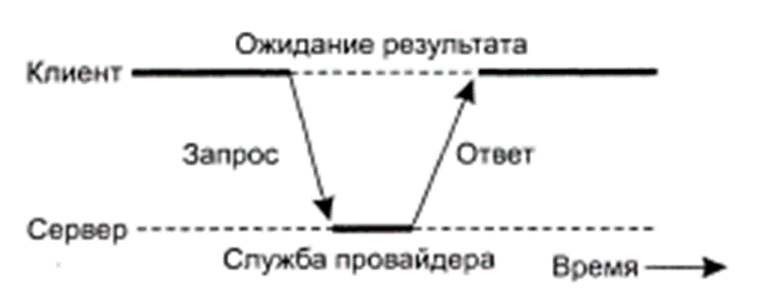
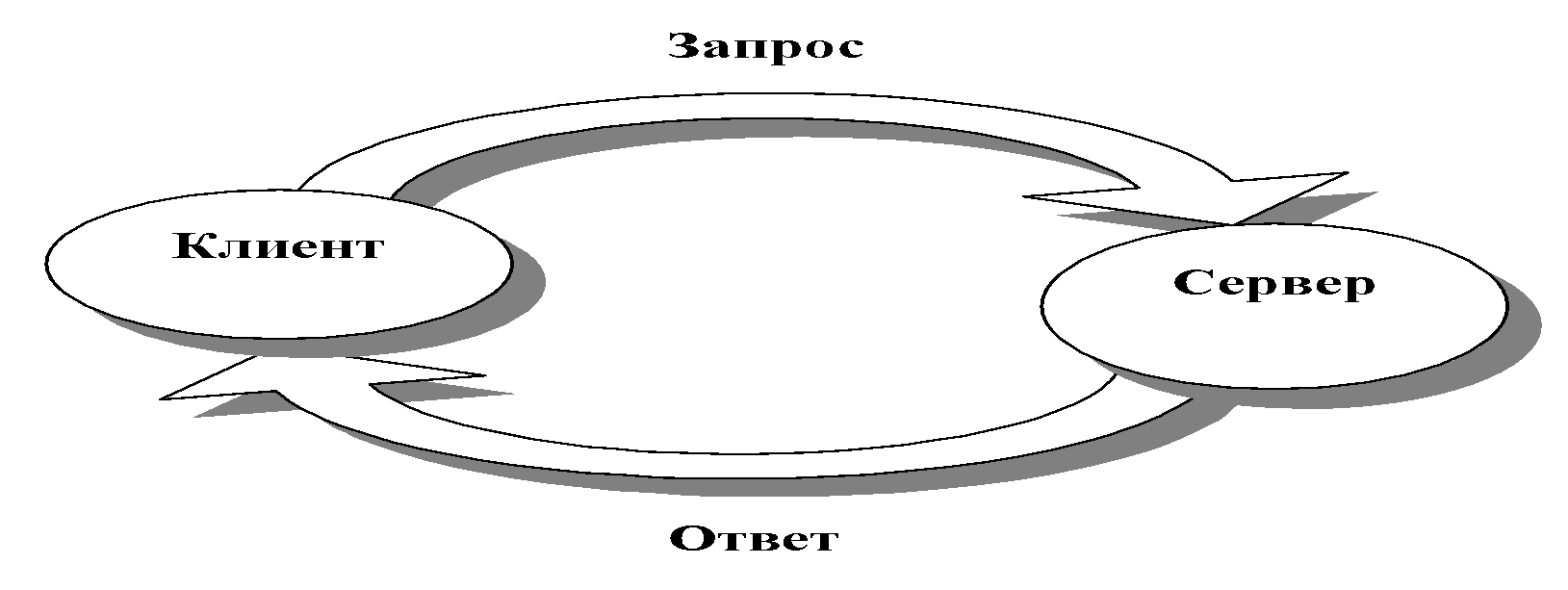
Э.Танненбаум описал следующие виды программных решений РИС:

* **Распределенная ОС** – это сильносвязанная ОС, которая используется для управления мультипроцессорными и гомогенными многомашинными системами. Обеспечивает сокрытие и управление аппаратным обеспечением.
* **Сетевые ОС** – это слабосвязанные ОС, предназначенные для управления гетерогенными вычислительными системами. Эти системы должны обеспечивать удаленный доступ к локальным службам.
* **ПО Промежуточного слоя** – это дополнительный уровень поверх сетевых операционных систем, реализующий службы общего назначения. Их задача обеспечение прозрачности распределения.

## **16. Модель клиент-сервер и трехуровневая организация приложений. Варианты архитектуры клиент-серверных систем**.

В классической архитектуре “клиент-сервер” любая информационная система должна иметь минимум три основные функциональные части:

* модуль хранения данных;
* модуль обработки;
* модуль интерфейса с пользователем.



В базовой модели клиент-сервер все процессы в распределенных системах делятся на две возможно перекрывающиеся группы. Процессы, реализующие некоторую службу, например службу файловой системы или базы данных, называются серверами {servers). Процессы, запрашивающие службы у серверов путем посылки запроса и последующего ожидания ответа от сервера, называются клиентами {clients).

**Трехуровневая организация**: Рассматривая множество приложений типа клиент-сервер, предназначенных для организации доступа пользователей к базам данных, многие рекомендовали разделять их на три уровня.

* уровень пользовательского интерфейса;
* уровень обработки;
* уровень данных.

**Три основных уровня корпоративных приложений**

* Представление – ПО, реализующее функции интерфейса пользователя - клиент
* Бизнес-логика приложения - ПО, реализующее функции автоматизируемых деловых процессов – (Сервер WEB + Сервер приложений)
* Источник данных - ПО, реализующее функции доступа к данным – сервер БД

**Варианты архитектуры к-с**

Простейшая организация предполагает наличие всего двух типов машин.

Клиентские машины, на которых имеются программы, реализующие только пользовательский интерфейс или его часть.

Серверы, реализующие все остальное, то есть уровни обработки и данных.

На самом деле такая система не является распределенной: все происходит на сервере, а клиент представляет собой не что иное, как простой терминал.

**Разновидности модели клиент-сервер**

* **Толстый клиент.** Такая модель подразумевает объединение в клиентском приложении интерфейса пользователя и обработки данных. Серверная часть в этом случае представляет собой сервер баз данных.
* **Тонкий клиент.** В этом случае клиентское приложение обеспечивает интерфейс с пользователем, а сервер объединяет модули хранения и обработки (толстый сервер).
* **Многоуровневые системы клиент-сервер.** В этих системах некоторая часть функций, связанных с обработкой данных, либо с доступом к модулям хранения, либо с обеспечением многопользовательского доступа, выделяется в отдельный модуль (mieddlware), называемый ПО промежуточного слоя.

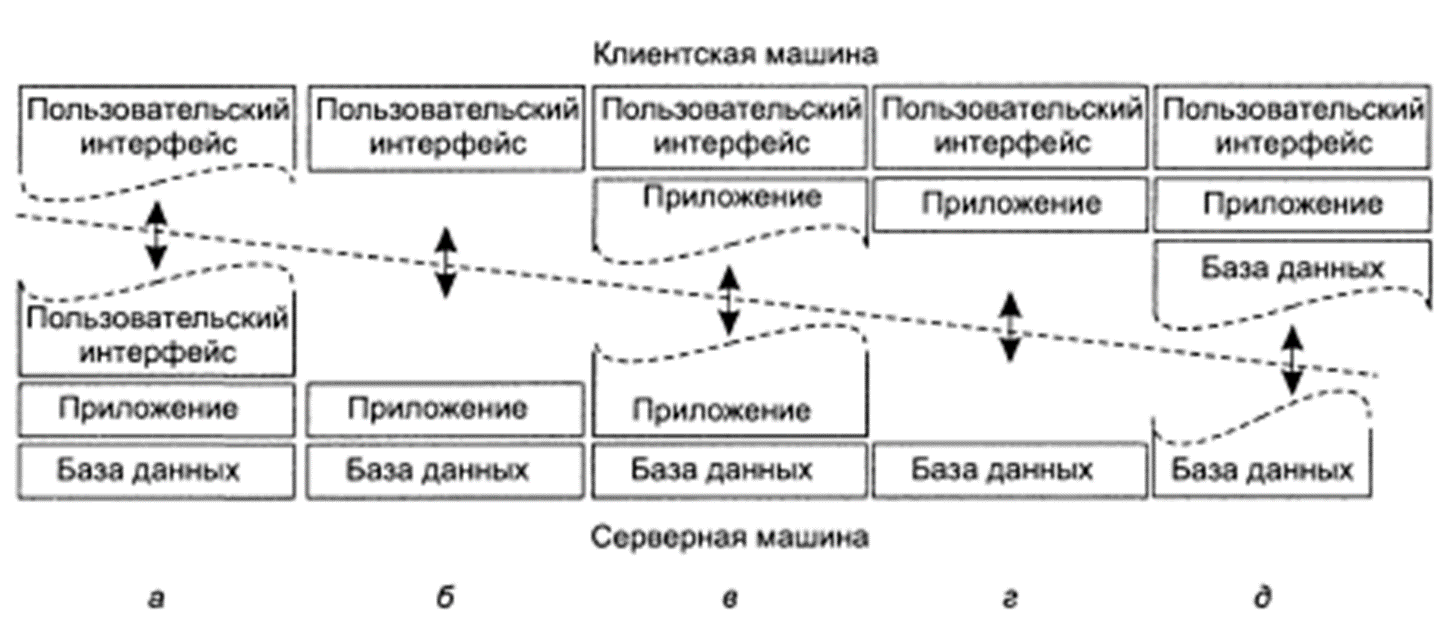
**Взаимодействие между клиентом и сервером**

Взаимодействие между клиентом и сервером расположенными на разных машинах часто называют «поведением запрос – ответ»

## **17. Физически двухзвенные и трехзвенные архитектуры приложений. Вертикальное и горизонтальное распределение программных компонентов РИС.**

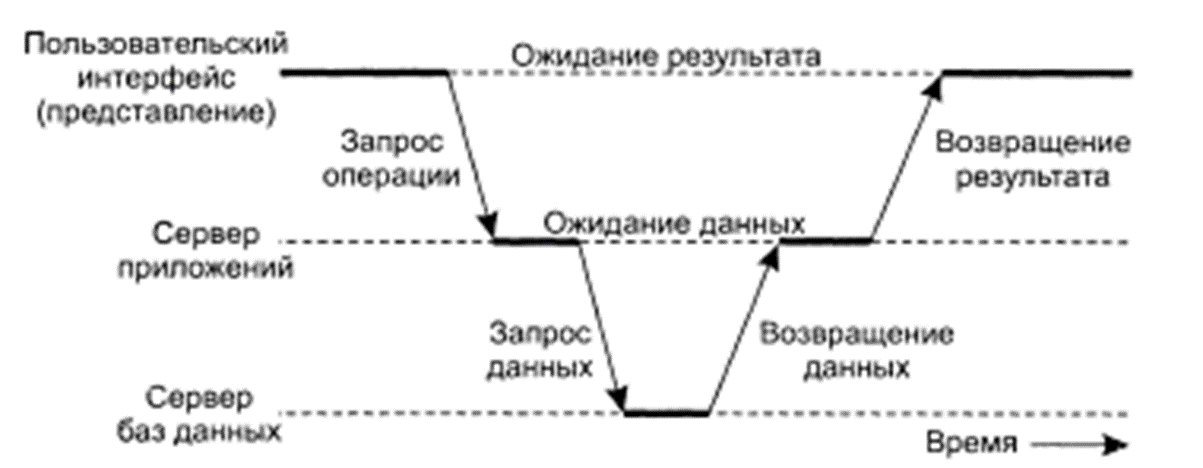
**Физически двухзвенные**

Один из подходов к организации клиентов и серверов — это распределение программ, находящихся на уровне приложений, по различным машинам.



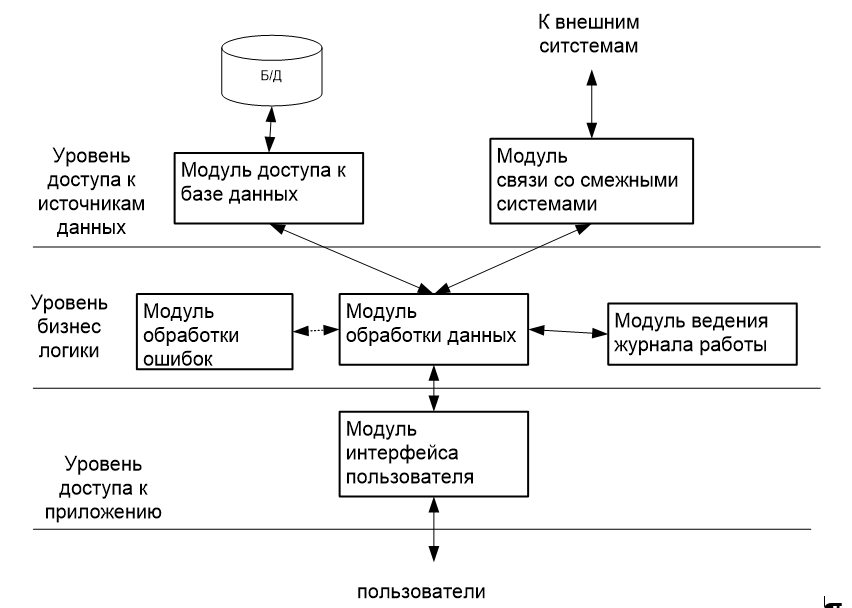
**Трехзвенные**

Рассматривая только клиенты и серверы, мы упускаем тот момент, что серверу иногда может понадобиться работать в качестве клиента.



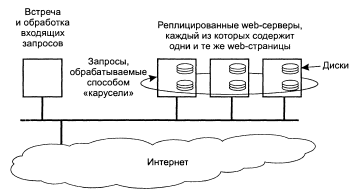
**Системы с вертикальным распределением**

Характеристической особенностью вертикального распределения является то, что оно достигается размещением логически различных компонентов на разных машинах



**Горизонтальное распределение**

* При таком типе распределения клиент или сервер может содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может происходить независимо. Это делается для выравнивания загрузки.
* Пример, web-сервер, peплициpoвaнный на несколько машин локальной сети.



# **Тема 3. Понятие Архитектуры РИС.**

## **18. Понятие ОРГАНИЗАЦИИ распределенной системы. Программная ОРГАНИЗАЦИЯ. Прозрачность РИС и ее Архитектура. Выбор варианта программной архитектуры. Системная архитектура. Виды системной архитектуры.**

Программное обеспечение РС распределяется между вычислительными узлами этой системы.

В организации систем часто выделяют:

* Логическую организацию совокупности программных компонент системы;
* Физическую организацию размещение этих компонент на узлах системы.

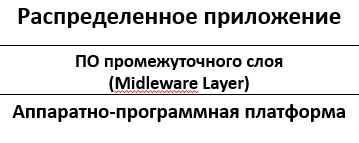
**Программная организация**   
Организация РС определяется составом программных компонент входящих в состав системы.

Программная организация показывает из каких программных компонентов состоит система, а также и то, как взаимодействуют между собой программные компоненты этой системы.

**Прозрачность**

Исходя из требования обеспечения прозрачности в распределенных системах требуется четко разделять приложения и лежащие в их основе платформы.

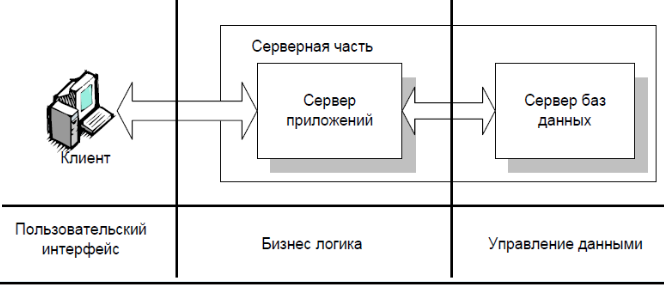
Такое разделение в РС выполняется с помощью промежуточного уровня системы.



**Выбор варианта**

Важнейшим решением при разработке архитектуры системы является выбор варианта размещения ПО промежуточного уровня (ППУ) системы.

Имеется различные методики определение состава и размещения ППУ приложений, что и определяет множество вариантов программных архитектур.

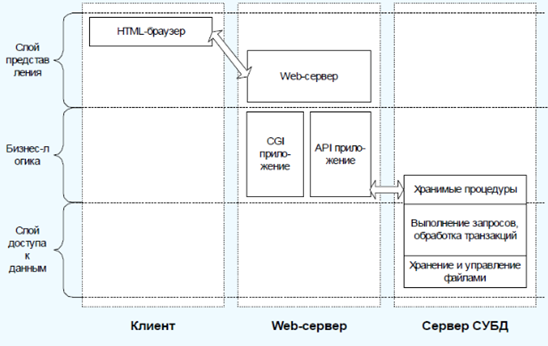


**Системная архитектура**

Фактическая (реально разворачиваемая) реализация РС, требует однозначного определения размещения программных компонент системы на реальных машинах.

Практически всегда имеется множество вариантов такого размещения.

Размещение программных компонент системы (программная архитектура) на физических машинах называется **системной архитектурой**.



**Виды архитектуры**

Различают три вида системной архитектуры:

* централизованная;
* децентрализованная (peer-to-peer);
* гибридная – комбинация элементов централизованной и децентрализованной архитектур.

## **19. Понятие архитектуры (архитектурного стиля) РИС. Понятие программного компонента. Основные виды архитектур РИС.**

При проектировании и создании РС выбор архитектуры является ключевым техническим решением, определяющим успех создания больших программных систем.

При обсуждении архитектурных аспектов РС важным понятием является **архитектурный стиль**, который описывается в терминологии компонент и определяет:

* способ коммуникаций между компонентами;
* порядок обмена данными между компонентами;
* как элементы системы совместно формируют распределенную систему.

**Понятие программного компонента**  
**Компонент** – модульная единица ПО снабженная полностью определенным и предоставляемым по запросу интерфейсом.

Компонент должен обладать свойством – заменяемости в рамках системного окружения. Замена компонента может быть выполнена в любой момент, даже в условиях работы системы.

**Интерфейс** - описывает состав параметров необходимых для обращения к компоненту. Замена компонента может быть выполнена только при условии неизменности его интерфейса.

**Коннектор** – это механизм который обеспечивает коммуникации, и способствует координации (или кооперации) компонент друг с другом. Коннектор может быть сформирован на основе средств, реализующих способ связи между компонентами:

* удаленный вызов процедур (RPC);
* обмен сообщениями (message passing);
* потока данных (streaming data) и д.р.

Использование понятий компонент и конектор позволяет описывать различные варианты конфигураций, которые в свою очередь могут быть квалифицированы как архитектурные стили.

**Виды архитектур РИС**

В настоящее время общепризнанными архитектурными решениями (стилями) считаются:

* многоуровневые архитектуры (layered);
* объектные архитектуры (object-based);
* компонентные архитектуры (component-based);
* сервисно-ориентированные архитектуры;
* ресурсо-центрированные архитектуры (resource-centerd);
* архитектуры, основанные на событиях (event-based).

Эти архитектурные решения могут одновременно применяться в одних и тех же системах в различных сочетаниях.

## **20. Характеристики распределенной ИС, зависящие от выбора архитектуры системы. Масштабируемость. Надежность. Доступность. Открытость. Гибкость. Безопасность**.

**Масштабируемость** – сохранение работоспособности системы при изменении масштаба системы.

* Масштабируемость системы может измеряться по трем показателям:
  + По размеру.
  + По географической распределенности.
  + По степени сохранения управляемости при увеличении масштаба РИС

Наиболее часто проблемы размера определяются заложенными при

ее создании централизацией: служб; данных; алгоритмов.

**Вертикальное масштабирование** – увеличение производительности каждого компонента системы с целью повышения общей производительности.

* возможность заменять в существующей вычислительной системе компоненты более мощными.

**Горизонтальное масштабирование** – возможность добавлять к системе новые узлы (серверы, процессоры) для увеличения общей производительности.

**Надежность** охватывает множество требований к распределенным системам, из которых важнейшими являются:

* доступность(availability);
* безотказность(reliability);
* безопасность(safety);
* ремонтопригодность(maintainability).

**Доступность** - свойство системы находиться в состоянии готовности к работе. Обычно доступность – это вероятность того, что система в данный момент времени будет правильно работать и окажется в состоянии выполнить свои функции, если пользователи того потребуют.

Система с высокой степенью доступности – это такая система, которая в произвольный момент времени, скорее всего, находится в работоспособном состоянии.

**Открытость** обеспечивает:

* интероперабельность - способность к взаимодействию с другими РИС
* переносимость – способность к переносу приложений между разными РИС.

Основывается на использовании при проектировании ИС открытых стандартов, спецификаций и протоколов, определяющих порядок взаимодействия между всех компонентов распределенной системы между собой, а также пользователей с распределенной системой.

**Гибкость** может быть следующих видов:

* Программной - замена программ (например, выпуск новых версий) и перенос их между хостами РИС
* Аппаратной - изменение числа и характеристик технических средств
* Структурной - изменение структуры РИС (например, переподчинение элементов, разбиение одной РИС на несколько, слияние нескольких РИС в одну)
* Логической - изменение бизнес-процессов (например, последовательности операций)

## **21. Архитектурные элементы как сущности распределенной ИС. Системный и проблемный аспекты выбора сущностей архитектуры РИС. Объекты, объектные компоненты, сервисы и ресурсы – как сущности архитектуры.**

**АРХИТЕКТУРА РИС определяет:**

* какие сущности входят в РС,
* каким образом они взаимодействуют.

Выбор сущностей и способы их взаимодействия в рамках РИС определяет весь спектр возможных архитектурных решений.

**Сущности**, образующие РИС, имеют два аспекта:

* системный;
* проблемный.

**Системная реализация сущности**

Системная сторона сущности определяет способ **физической реализации функций сущности** и методов ее **взаимодействия** с другими сущностями.

Исходя из системной стороны сущностей, взаимодействие между ними рассматривается как модель **межпроцессного** взаимодействия между распределенными сущностями.

При этом помимо классического процесса имеются следующие варианты их реализации:

* В примитивных системах, например, таких как системы датчиков, ОС узлов системы могут не поддерживать механизм процессов, в этом случае такие сущности называют **узлами**;
* В большинстве современных ОС абстракция процесса дополняется абстракцией **потока**. Потоки, как и процессы могут обслуживать **конечные точки подключений**.

Точка подключения: - сетевой адрес узла/процесса/потока. Пример: URL/URI

**Проблемная реализация сущности**

Определяется предметной областью, для автоматизации которой используются сущности РС.

Предметная сторона определяет:

* алгоритмы обработки информации;
* порядок обмена информацией между сущностями в рамках РС.

Наиболее часто в качестве проблемно-ориентированной абстракции сущностей РИС используются:

* объекты; компоненты; службы; ресурсы.

**Объекты**

Выбор объектов в качестве сущностей РИС обусловлен во многом объектно-ориентированным подходом в программировании, а также к проектированию РИС.

**Объекты РС** представляют собой абстракцию объектов реального мира, относящихся к конкретной проблемной области.

Разделение предметной области на объекты выполняется исходя из естественной декомпозиции предметной области на ряд взаимодействующих друг с другом объектов.

Доступ к объектам реализуется через их интерфейсы, которые описываются с помощью языка IDL-Interface Definition Language, обеспечивающего описание методов, определенных для объекта.

**Сервисы**

Сервисы являются сущностями близкими к объектам и компонентам, также использующими подход, основанный на инкапсуляции поведения в компонент и использовании интерфейса для получения доступа друг к другу.

**Web-сервисы**

В отличие от первых двух видов сущностей web-сервисы в действительности интегрированы в web и используют стандарты www для представления и описания сервисов.

W3C определил **Web-сервисы** как: Программное приложение, идентифицируемое с помощью URI, чей интерфейс и связывание реализуется на основе использования XML.

То Web-сервисы в целом выглядят как полноценные сервисы, которые могут при их комбинировании образовывать сложные сервисы с дополнительными возможностями.

**Ресурсы**

Одной из проблем построения РИС на основанных на Web сервисах явилась высокая сложность обеспечения связи между большим числом различных компонент.

В качестве альтернативы было предложено рассматривать РИС как коллекцию ресурсов каждый из которых индивидуально управляется своим компонентом (сервисом).

## **22. Модели коммуникаций при взаимодействии между сущностями архитектуры РИС: Межпроцессные коммуникации (IPC – Interprocess communications). Удаленный вызов. Непрямые коммуникации (indirect communications). Групповые коммуникации.**

Имеется **три вида** прямых коммуникаций между сущностями РС:

* **Межпроцессные коммуникации (прямые коммуникации);**

Этот вид коммуникаций относится к низкоуровневому способу поддержки взаимодействия между процессами выполняющимися различными сущностями в распределенной системе.

К этому типу коммуникаций относятся следующие типы:

* + Обмен сообщениями (message passing);
  + Прямой доступ к API протоколов Internet (программирование сокетов);
  + Групповые коммуникации.
* **Удаленный вызов компонент;**

Представляет собой способ коммуникаций, наиболее широко используемый в распределенных системах.

Объединяет ряд методов, основанных на двустороннем обмене информацией между сущностями:

* + - Запрос-ответ.
    - Вызов удаленных процедур (RPC).
    - Удаленный вызов методов (RMI).
* **Непрямые коммуникации.**

К непрямым коммуникациям относятся способы взаимодействия между двумя сущностями посредством третьей стороны, что уменьшает степень связности между передатчиком и приемником. Такие варианты взаимодействия возникают, например когда:

* + - передающей стороне не требуется знать, кто является получателем (пространственное разъединение);
    - передающей и принимающей сторонам не требуется находиться в рабочем состоянии в одно и то же время (временное разъединение).

**Групповые коммуникации**

Эта модель предполагает доставку сообщений набору получателей. Соответствует парадигме один-ко-многим. Групповая модель взаимодействия основана на абстракции группы, представленной в системе идентификатором группы.

Получатель подключается к группе путем отправки сообщения о присоединении к группе. Отправители отправляют сообщения в группу по ее идентификатору и не знают конкретный состав получателей.

Группы, как правило, обслуживают членство в группе, для чего имеют механизм выхода из группы.

## **23. Модели коммуникаций при взаимодействии между сущностями архитектуры РИС: Системы публикации-подписки (publish-subscribe). Очередь сообщений (Message queues). Пространство записей (кортежей) (Tuple spaces). Распределенная разделяемая память (Distributed Shared Memory).**

**Модели непрямых коммуникаций**

К этому виду моделей относятся:

* + групповые коммуникации;
  + системы публикации-подписки;
  + очереди сообщений;
  + кортежное пространство;
  + распределенная разделяемая память.

**Системы публикации-подписки (publish-subscribe)**

Примером таких систем могут служить системы торговли различными ресурсами, в которых большое число производителей (публикаторов) распространяют информацию среди большого числа потребителей (подписчиков), для таких систем не подходят модели прямых коммуникаций.

Системы публикации-подписки содержат промежуточный сервис, который является ключевым элементом этих систем.

Этот сервис должен обеспечивать гарантированное распространение информации, генерируемой публикаторами ко всем подписчикам, желающим получать данную информацию.

**Очередь сообщений (Message queues)**

В то время как системы публикации-подписки поддерживают стиль один-ко-многим, очереди сообщений предлагают модель обслуживания один-к-одному (точка-точка).

В этом случае процесс-производитель сообщений отправляет их в определенную очередь, а процесс-потребитель может получать сообщения из этой же очереди либо получать уведомления о поступлении в очередь новых сообщений.

**Пространство записей (кортежей) (Tuple spaces)**

Пространство кортежей (Tuple spaces) предполагает, что процесс-отправитель может поместить некоторый произвольный экземпляр структурированных данных (кортеж) в постоянное пространство записей, а другие процессы могут либо читать, либо удалять такие записи с помощью шаблонов, определяющих интересующую их информацию.

Так как пространство записей является постоянным, процессы "читатели" и процессы "писатели" не должны существовать в одно и то же время.

**Распределенная разделяемая память (Distributed Shared Memory)**

Представляет собой абстракцию разделения данных между процессами, которые не разделяют между собой физическую память.

Тем не менее, программисты получают возможность писать/читать структуры данных также легко, как если бы данные находились в собственном адресном пространстве процесса.

## **24. Роли и ответственность сущностей РС. Архитектура клиент-сервер. Архитектура Peer-to-Peer.**

**Роли и ответственность сущностей РС**

В распределенной системе процессы, представленные соответственными объектами, компонентами или сервисами (включая Web-сервисы), взаимодействуют друг с другом, выполняя необходимые действия, в рамках выполняемых ими функций (ролей), например, поддержка сессии чата.

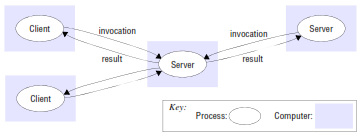
Роль, выполняемая сущностью, является фундаментом для определения архитектуры системы.

Имеется два вида архитектурных систем, определенных **на основе ролей**, выполняемых элементами РС:

* + клиент-сервер;
  + peer-to-peer.

**Клиент-сервер**

Эта архитектура получила очень широкое распространение.



На рисунке приведено взаимодействие между сущностями в системе DNS

**Архитектура Peer-to-Peer**

В этой архитектуре все процессы участвуют в работе на основе одной общей для всех роли, которая не накладывает никаких ограничений, связанных с выполнением ролей клиента или сервера.

Все сущности используют одно и тоже программное обеспечение и предоставляют один и тот же интерфейс для взаимодействия с другими сущностями.

## **25. Стратегии размещения сущностей распределенной ИС. Мультисерверная архитектура сервисов. Кэширование данных. Мобильный код. Мобильные программные агенты**.

**Стратегии Размещения сущностей**

Определяет каким образом логические сущности системы соотносятся с физической инфраструктурой системы, которая может содержать большое число машин, связанных между собой сетью произвольной сложности.

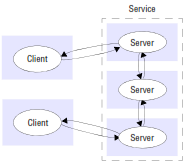
Размещение критически влияет на такие свойства распределенных систем как: Производительность; Надежность; Безопасность.

Вопрос размещения ПО заданного клиента или сервера на конкретных машинах, а также определение процессов выполняемых в каждой из машин распределенной системы является предметом разработки архитектуры РС. Размещение требует знания требований к размещению со стороны приложений.

Имеется несколько хорошо известных стратегий размещения:

* + Отображение служб на несколько серверов (mapping of services to multiple servers);
  + Кэширование;
  + Использование мобильного кода;
  + Использование мобильных программных агентов.

**Мультисерверная архитектура сервисов**



Службы могут быть реализованы как несколько серверных процессов, выполняемых в рамках одного или нескольких физических серверов. Примеры:

* + Каждый Веб-сервер управляет своими собственными ресурсами. Браузер может перемещаться между серверами получая доступ к ресурсам каждого из них.
  + При обработке запросов пользователей в системе, построенной на основе сервисов, при обработке запроса клиента один сервис может обращаться к другим сервисам для обслуживания. В этом случае говорят о сложных сервисах.

**Кэш** – это хранилище последних данных, полученных в результате взаимодействия между сущностями РС. При очередном обращении к сущности полученные в результате данные добавляются в это хранилище.

Ярким примером использования кэширования является кэширующий веб прокси сервер.

Примером **мобильного** кода являются JavaScript (VBScript), Java апплеты. Использование мобильного кода позволяет перенести выполнение действий с сервера на клиент. Позволяет улучшить время реакции, но требует дополнительных ресурсов со стороны клиента.

**Мобильный программный агент** – это исполняемая программа (код + данные), которая перемещается между машинами распределенной системы по сети и в конечном итоге возвращается с результатом.

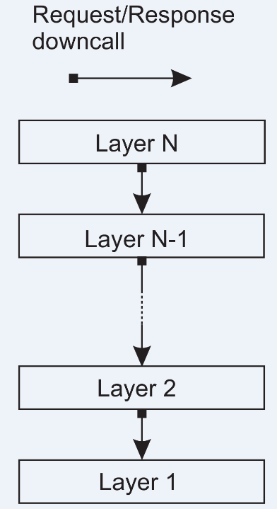
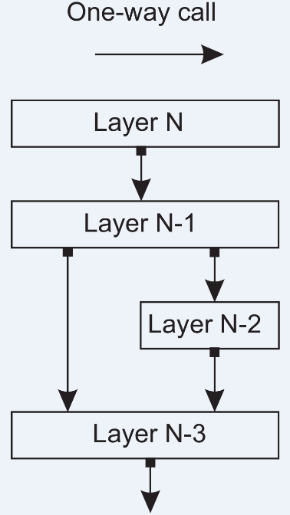
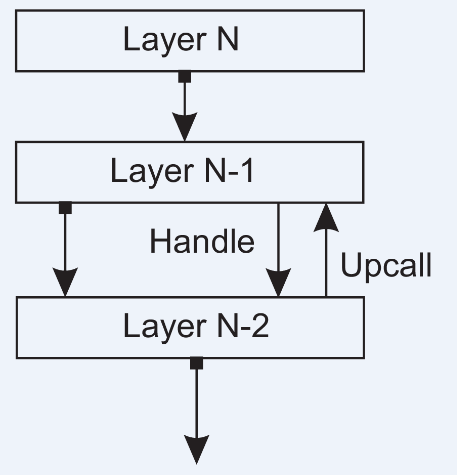
Мобильный агент может выполнять много обращений к локальным ресурсам машины, которую он посещает. По сравнению с удаленными обращениями к ресурсам применение мобильных агентов позволяет уменьшить стоимость коммуникаций и время получения нужных данных.

Использование мобильных агентов таит в себе опасности нарушения безопасности ресурсов машин, которые они посещают.

# **Тема 4. Многоуровневые архитектуры ПО промежуточного слоя**

## **26. Многоуровневая архитектура. Смешанная архитектура и архитектура с обратными связями. Расслоение и многоуровневая организация классического приложения.**

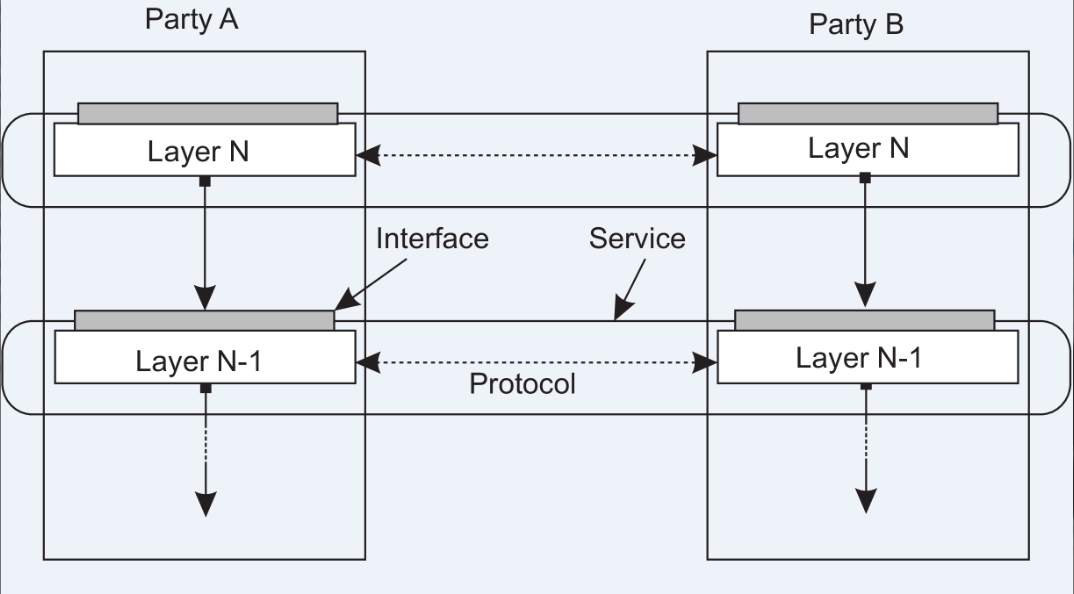
Простая многоуровневая архитектура/ Смешанная Многоур. архитектура/ Архитектура с обратными связями

Базовая идея проста: компоненты распределяются по уровням, при этом компонент уровня j Lj может вызывать компонент, находящийся на нижележащем уровне i (Li), от которого он получает ответ. Как правило вызов компонента вышестоящего уровня нижележащим запрещен.

Смешанная многоуровневая архитектура – допускает вызов не только смежных нижележащих уровней, но и более глубоколежащих уровней.

**Многоуровневая организация**

Примером такой организации может служить стек сетевых протоколов, например: TCP/IP

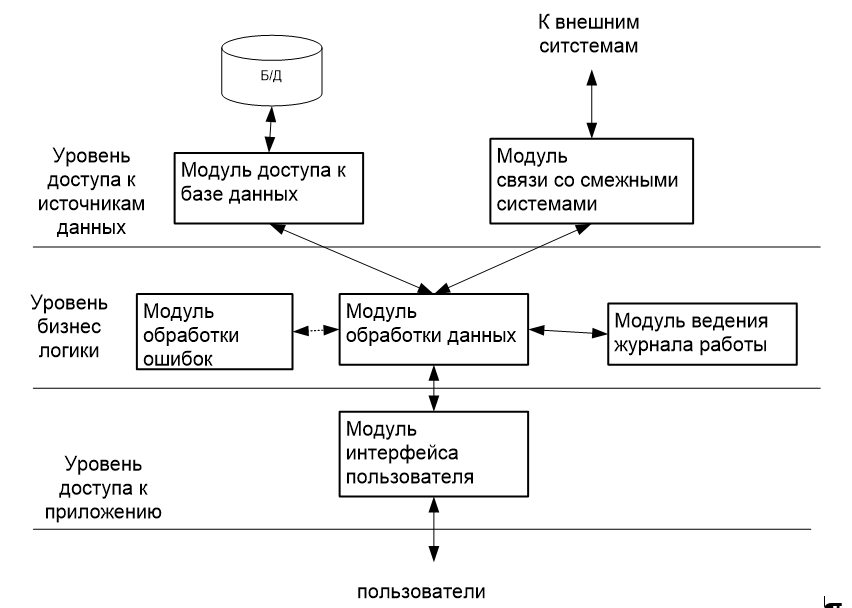
В этой архитектуре определены следующие понятия (концепты):

интерфейс - определяет функции нижележащего уровня, вызываемые вышележащим уровнем;

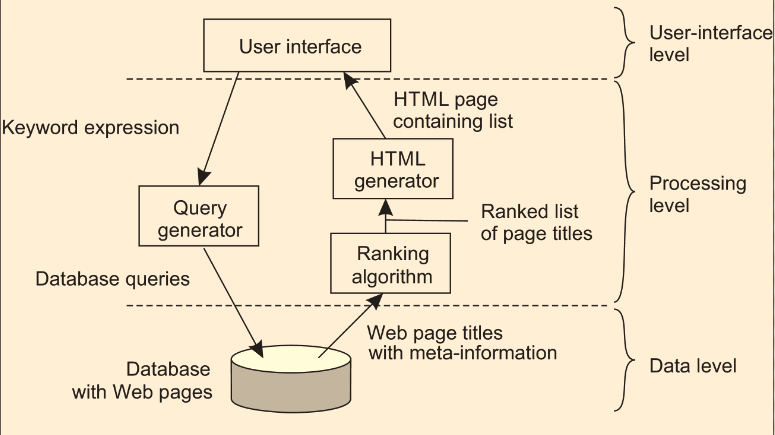
протокол - процедуры и правила, которым должны следовать обе взаимодействующие стороны при обмене информацией на данном уровня;

сервис, реализуется средствами одного уровня.

**Архитектура приложения (расслоение приложения)**



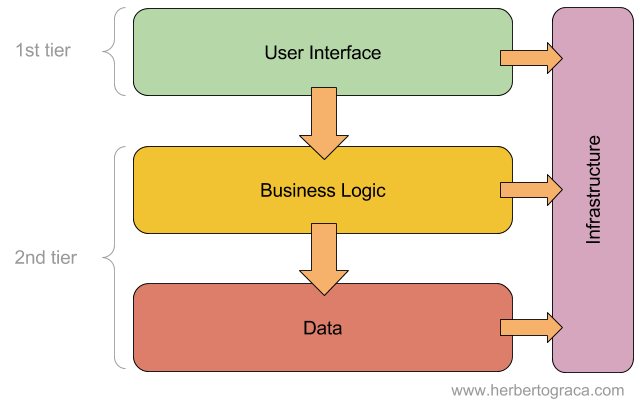
**Многоуровневая организация приложений**



## **27. Эволюция многоуровневой архитектуры приложений. Терминальные приложения. Корпоративные приложения. Корпоративные Web-приложения. Архитектура систем, управляемых бизнес-логикой.**

* Терминальные приложения 1960-1970-е были, по сути, одноуровневыми.
* 1980-1990. Корпоративные приложения

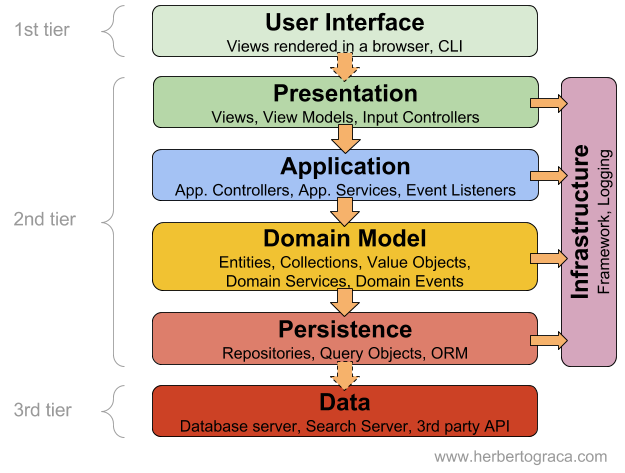
С появлением технологий б/д сформировались три классических уровня



* Середина 1990-х. Корпоративные Web-приложения

Уровень приложения разделился на подуровни:

* + Представления;
  + Сервер приложений;
  + Модель данных;
  + Перманентность объектов.

****

* Начало 2000-х. архитектура систем, управляемых бизнес логикой
* User Interface – уровень взаимодействия с пользователем, где пользователем может быть не только человек, но и программа.
* Aplication – уровень приложения, выполняет координацию (оркестровку) объектов предметной области, относящихся к уровню приложения.
* Domain – уровень бизнес логики, сущностей, событий и других объектов. **Является сердцем системы**.
* Infrastructure - поддержка перманентных объектов и обмен сообщениями.

## **28. Программные и аппаратные уровни распределенных ИС. Архитектурные шаблоны многоуровневых систем: - Многозвенная) архитектура; тонкий клиент и технология VNC; шаблон с использованием прокси; шаблон на основе брокера; шаблон на основе рефлексии.**

**Программные и аппаратные уровни в РС**

Примеры существующих платформ:

* Intel x86/Windows;
* Intel x86/Mac OS;
* Intel x86/Linux;
* ARM/Symbian;
* ARM/Linux и т.д.

Платформа обеспечивает обслуживание вышестоящих уровней, реализуется независимо на каждом компьютере РС и предоставляет программный интерфейс к функциям вычислительных процессов, коммуникаций и координации между процессами, выполняемыми на различных узлах системы. Она состоит из программного и аппаратного уровней:

* + Уровень ОС;
  + Уровень аппаратного вычислительного и сетевого обеспечения.

**Архитектурные шаблоны**

Архитектурные шаблона основаны на совместном использовании примитивных архитектурных элементов (сущностей, типов коммуникаций, ролей, размещений и др.).

Имеется множество архитектурных шаблонов, среди которых, наиболее часто используемыми являются те, что в вопросе

**Многоярусная (многозвенная) архитектура**

Эта архитектура является архитектурой, дополняющей многоуровневую архитектуру.

В то время как многоуровневая архитектура относиться к вертикальной организации сервисов в рамках абстракции уровней, ярусная архитектура является способом организации данного уровня и размещения этой функциональности на серверах, соответствующих физических машин.

**Тонкий клиент**

Термин тонкий клиент означает, что уровень пользовательского интерфейса располагается на клиенте, а исполняемый код, реализующий основную функциональность приложения, располагается на сервере (удаленном компьютере).

Достоинством такого подхода является возможность использование в качестве клиента простых локальных устройств, например, смартфонов и других устройств с ограниченными ресурсами.

Этот шаблон характерен для облачных вычислений.

**VNC (Virtual Network Computing)**

Концепция тонкого клиента привела к появлению технологии VNC - вычисление виртуальных сетей.

Начальные идеи VNC эволюционировали, в результате чего появилась реализация RealVNS, а затем TightVNC, Apple Remote Desktop и др.

**Шаблон Proxy**

В этом шаблоне прокси создается в локальном адресном пространстве для представления удаленного объекта. Этот посредник предоставляет в точности тот же интерфейс, что и удаленный объект.

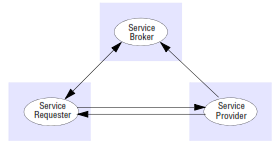
Использование шаблона на основе посредника обеспечивает поддержку локальной прозрачности в RMI и RPC коммуникациях.

Прокси могут использоваться также для инкапсуляции других видов функциональности, например:

* + Размещения политик;
  + Обеспечения репликаций;
  + Выполнения кэширования.

**Использовано брокера**

В случае использования в качестве сущности сервиса часто используется архитектура, обеспечивающая взаимодействие между сущностями с помощью брокера сервиса.

****

**Рефлексия**

Эта модель (шаблон) все чаще используется в распределенных системах как средство обеспечения:

1. Динамического обнаружения свойств системы (интроспекция);

2. Реализация способности динамического изменения ее структуры или поведения (интерцессия).

В рефлексивных системах на базовом уровне доступен стандартный интерфейс сервиса, а на мета-уровне становятся доступными компоненты и их параметры, задействованные (участвующие) в реализации сервиса.

# **Тема 5. Архитектурные решения для ПО промежуточного слоя**

## **29. Назначение ПО промежуточного слоя. Категории ПО промежуточного слоя. Примеры РИС на основе различных категорий ПО промежуточного слоя.**

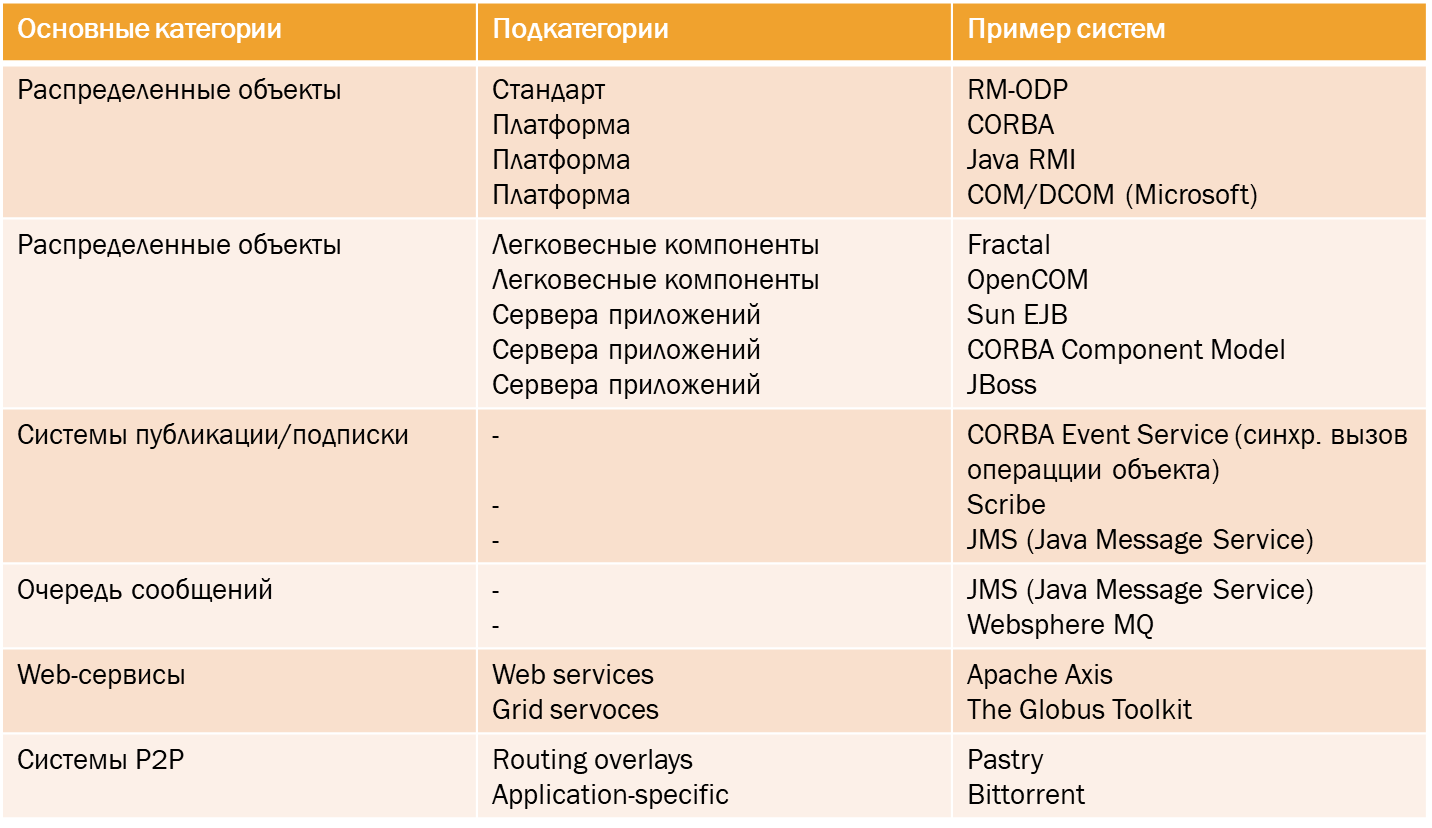
**Назначение ПО промежуточного слоя (ППС)**

Задача промежуточного ПО состоит в представлении высокоуровневой программной абстракции для разработки РС.

Это ПО благодаря использованию дополнительного уровня позволяет абстрагироваться от гетерогенности нижележащих инфраструктур с целью обеспечения интероперабельности и переносимости.

Промежуточное ПО изолирует логику приложений от уровня сетевого взаимодействия и ОС

Также как и архитектурные шаблоны, категория ППС определяется на основе выбора вида сущностей РС и коммуникационной парадигмы (модели). Приведем наиболее распространенные категории ППС:



## **30. Примеры применения ПО промежуточного слоя в распределенных системах. Мониторы обработки транзакций. Промежуточное ПО, ориентированное на обработку сообщений (Message Oriented Middleware - MOM). Корпоративный сервер приложений. Ограничения ПО промежуточного слоя. Ограничения ПО промежуточного слоя.**

**Мониторы обработки транзакций**

Первоначально основной задачей ТР-монитора в среде клиент-сервер было сокращение числа соединений клиентских систем с базами данных. При непосредственном обращении клиента к серверу базы данных для каждого клиента устанавливается соединение с СУБД, которое порождает запуск отдельного процесса UNIX. ТР-мониторы брали на себя роль концентратора таких соединений, становясь посредником между клиентом и сервером базы данных.

Каждая транзакция – это законченный блок обращений к ресурсу (как правило, базе данных) и некоторых действий над ним, для которого гарантируется выполнение четырех условий, так называемых свойств ACID (Atomicity, Consistency, Isolation, Durability).

**Промежуточное ПО, ориентированное на обработку сообщений (Message Oriented Middleware - MOM)**

МОМ реализует скорее равноправные (peer-to-peer), чем подчиненные (клиент-сервер) отношения между модулями приложениями. МОМ можно считать и наиболее гибкой категорией MW, поскольку системы этого типа поддерживают как синхронные, так и асинхронные коммуникации на базе сетевых протоколов с установлением и без установления соединения. По способу обмена сообщениями все продукты МОМ разбиваются на три подгруппы:

* + передача сообщений,
  + очереди сообщений,
  + подписка/публикация.

Большая часть МОМ-систем реализует асинхронный механизм очередей сообщений. В отличие от передачи сообщений, эта модель межпрограммного взаимодействия не требует поддержки непосредственного соединения одного прикладного модуля с другим, но гарантирует доставку сообщения даже в том случае, если программа-адресат в данный момент не доступна.

Программа-отправитель передает сообщение MQ-системе и продолжает выполнение. Сообщение помещается в локальное промежуточное хранилище - очередь сообщений, размещенную в оперативной памяти или на диске, откуда оно может быть немедленно или через какое-то время передано программе-получателю

**Корпоративный сервер приложений**

Появление серверов приложений как отдельных готовых решений связано и с бурным вторжением Web в сферу корпоративных высококритичных систем. Серверы приложений являются одной из ключевых составляющих IT-инфраструктуры значительной части современных крупных предприятий.

Примеры серверов приложений от различных поставщиков: **oracle, sql**

**Ограничения ПО промежуточного слоя**

* Снижение производительности системы в целом за счет исполнения промежуточного ПО (на сервере или клиенте).
* Замедление сетевого обмена между клиентом и сервером данных, за счет работы сервера приложений.
* Возможность появления новых уязвимостей в безопасности системы, за счет возможной уязвимости промежуточного ПО.
* Снижение надежности системы за счет включения дополнительного промежуточного ПО.

# **Тема 6. Фундаментальные модели РИС.**

## **31. Понятие фундаментальной модели РИС. Три фундаментальные модели РС.**

Все рассмотренные архитектурные модели основываются на следующих фундаментальных свойствах РИС:

* + Обеспечение надежных коммуникаций между процессами, исполняющимися на различных узлах системы;
  + Обеспечение надежной работы и высокой производительности процессов, реализующих функции системы;
  + Обеспечение безопасного состояния всех ресурсов распределенной системы.

Исходя из этих свойств можно определить три фундаментальных модели распределенной системы:

* + Модель взаимодействия в распределенной системе;
  + Модель отказоустойчивости распределенной системы;
  + Модель информационной безопасности распределенной системы.

## **32. Архитектуры, основанные на объектах и объектных компонентах.**

В этой архитектуре понятие **объект** однозначно соответствует понятию **компонент**. Объекты взаимодействуют между собой с помощью механизма **RPC.** Эта архитектура предоставляет естественный способ инкапсуляции данных (**состояние объекта**) и операций которые могут выполняться над этими данными (**методы объекта**) объектом.

**Интерфейс,** предоставляемый объектом, скрывает детали реализации объекта, что делает объект полностью независимым от его окружения.

Разделение интерфейсов и реализации объектом этого интерфейса позволяет разместить интерфейс объекта на одной машине, а сам объект на другой машине. Такой объект называется **распределенным**.

**Распределенные объекты**

Когда клиент связывается с распределенным объектом, реализация интерфейса этого объекта, называемая **proxy**, загружается в адресное пространство клиента. Посредник (proxy) это аналог заглушки (stub) клиента в RPC. Он выполняет маршалинг вызова метода в сообщение и извлечение результата из ответного сообщения, получаемого от сервера. Реальный объект располагается на удаленной машине, где имеется такой же интерфейс как и на клиентской машине. Входящий вызов метода принимает серверная заглушка, которая извлекает вызов из сообщения и передает его интерфейсу объекта сервера. Заглушка сервера размещает ответ в сообщение и отправляет его proxy клиента

## **33. Архитектура, основанная на сервисах.**

Архитектура распределенных объектов позволяет на ее основе сформировать **сервис** как независимую программную единицу.

Сервис реализуется как **самодостаточная сущность,** созданная на основе распределенного объекта. Понятие сервиса как независимой самостоятельной единицы РИС, позволяет определить сервис-ориентированную архитектуру.

**Основные понятия Сервис Ориентированной Архитектуры (1)**

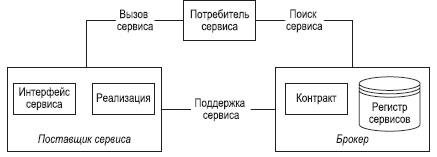
* **СOA** – это способ описания требований и методология предоставления сервисов, независимых от программных платформ и языков разработки, для использования при создании распределенных приложений.
* **Сервис** – это повторно-исполняемая задача в рамках бизнес-процесса.
* **Процесс** (бизнес задача) – это композиция, составленная из отдельных сервисов.
* Процесс определяет логику взаимодействия сервисов, независимо от их реализации.

**Основные понятия СОА (2)**

* SOA описывается как совокупность сервисов, реализуемых в виде компонентных объектов, систематизированных на основе обмена сообщениями.
* Общим примером сообщений, которыми обмениваются сервисы является XML сообщение.
* Для каждого сервиса принято определять:
  + поставщика сервиса (компонент сервис),
  + потребителя сервиса (компонент клиент);
  + брокера - компонент, обеспечивающий взаимодействие поставщика и потребителя.

**Взаимодействие между поставщиком и потребителем сервиса**

Клиент обращается к поставщику посылая ему сообщение стандартного формата, содержащее метаданные на основе которых поставщик и будет действовать. Поставщик на основе полученных в сообщении метаданных формирует ответ. Ответ принятый от поставщика клиент может использовать по своему усмотрению.



**Архитектура ИС предприятия на основе SOA**

* Уровень бизнес-логики предприятия
* Уровень логики приложения
* Уровень приложения

## **34. Принцип слабой связи в РИС на основе SOA. Достоинства слабой связи. Понятие состояния сервиса.**

В определение SOA входит понятие слабой связи сервисов. Этот термин подразумевает, что взаимодействующие программные компоненты имеет минимальное знание друг о друге: *они находят информацию, которая им нужна для взаимодействия непосредственно перед взаимодействием.*

***Достоинства слабой связи***

* ***Гибкость:*** *сервис может быть расположен на любом сервере, а при необходимости* - перемещен. Пока ссылка на этот сервис есть в службе регистрации, предполагаемые клиенты будут в состоянии найти и использовать его.
* ***Масштабируемость:*** *функциональные возможности сервиса могут быть расширены* или сужены, поскольку при этом описание сервиса динамически меняется и, соответственно, изменяются и запросы.
* ***Возможность модификации реализации:*** *при условии, что оригинальные* интерфейсы сохраняются, реализация сервиса может быть обновлена без сбоев в обслуживании клиентов.
* ***Отказоустойчивость:*** *если возникают проблемы в работе сервера, программного* компонента или сегмента сети, или сервис становится недоступным по любой другой причине, клиенты могут сделать запрос к службе регистрации для обнаружения другого сервиса, который предоставляет требуемые услуги.

**Состояние сервиса**

в теории слабосвязанных сервисов важными являются понятия:

* + ≪состояние сервиса≫;
  + ≪сервис без состояний≫.

Для сложных запросов, которые требуют нескольких шагов, сервис сохраняет в своей локальной памяти некоторую информацию ("состояние") о первом шаге, предполагая использовать ее, когда клиент входит с ним в контакт на следующем шаге. В этом случае, говорят, что сервис обладает состоянием, и клиент должен возвратиться к тому же самому сервису при следующем шаге.

## **35. Архитектура на основе Веб-сервисов. Модель работы SOAP/XML Web-сервиса. Стек протоколов web-сервиса.**

**Веб-сервисы (или веб-службы)** – это распределенные программные компоненты, идентифицируемые своим сетевым адресом, интерфейс которых описан на специальном ≪диалекте≫ языка XML, а именно WSDL (Web Service Description Language).

Другие программные системы могут взаимодействовать с веб-сервисами согласно этому описанию посредством сообщений, основанных на другом ≪диалекте≫ XML - SOAP, и передаваемых с помощью интернет-протоколов

SOA и веб-сервисы являются ≪ортогональными≫ понятиями:

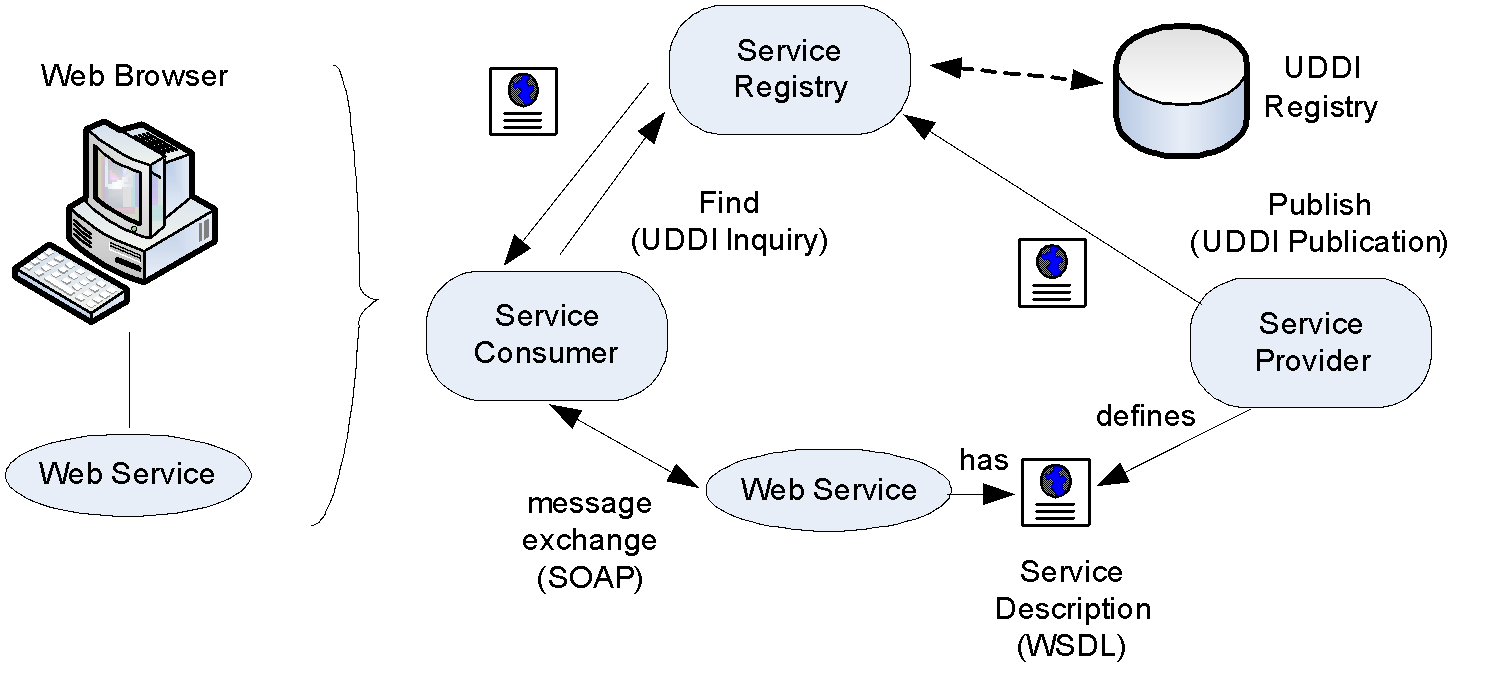
* + сервисная ориентация – это архитектурный стиль построения системы,
  + веб-сервисы – технология реализации сервиса.

Они, конечно, могут использоваться совместно – как это часто и случается, но они взаимно независимы.

Веб-сервисы хорошо подходят в качестве строительных блоков SOA-среды

**Веб-служба**, *веб-сервис* (англ. *web service*) — идентифицируемая программная система со стандартизированными интерфейсами.

**Модель работы Web-сервиса**



**Стек протоколов web-сервиса**



## **36. Архитектура, основанная на ресурсах. Ресурсо-центрическая распределенная система. Web-сервисы RESTful.**

Одной из проблем построения РИС на основанных на Web сервисах явилась высокая сложность обеспечения связи между большим числом различных компонент.

В качестве альтернативы было предложено рассматривать РИС как коллекцию ресурсов каждый из которых индивидуально управляется своим компонентом (сервисом).

**Ресурсо-центрическая архитектура**

Ресурсы могут добавляться либо удаляться с помощь приложения (удаленного). Такой подход привел формированию понятия ресурсо-центрической архитектуры РИС. Ресурсный подход получил широкое распространение в WEB в виде Web-сервисов RESTful

**Web-сервисы RESTful**

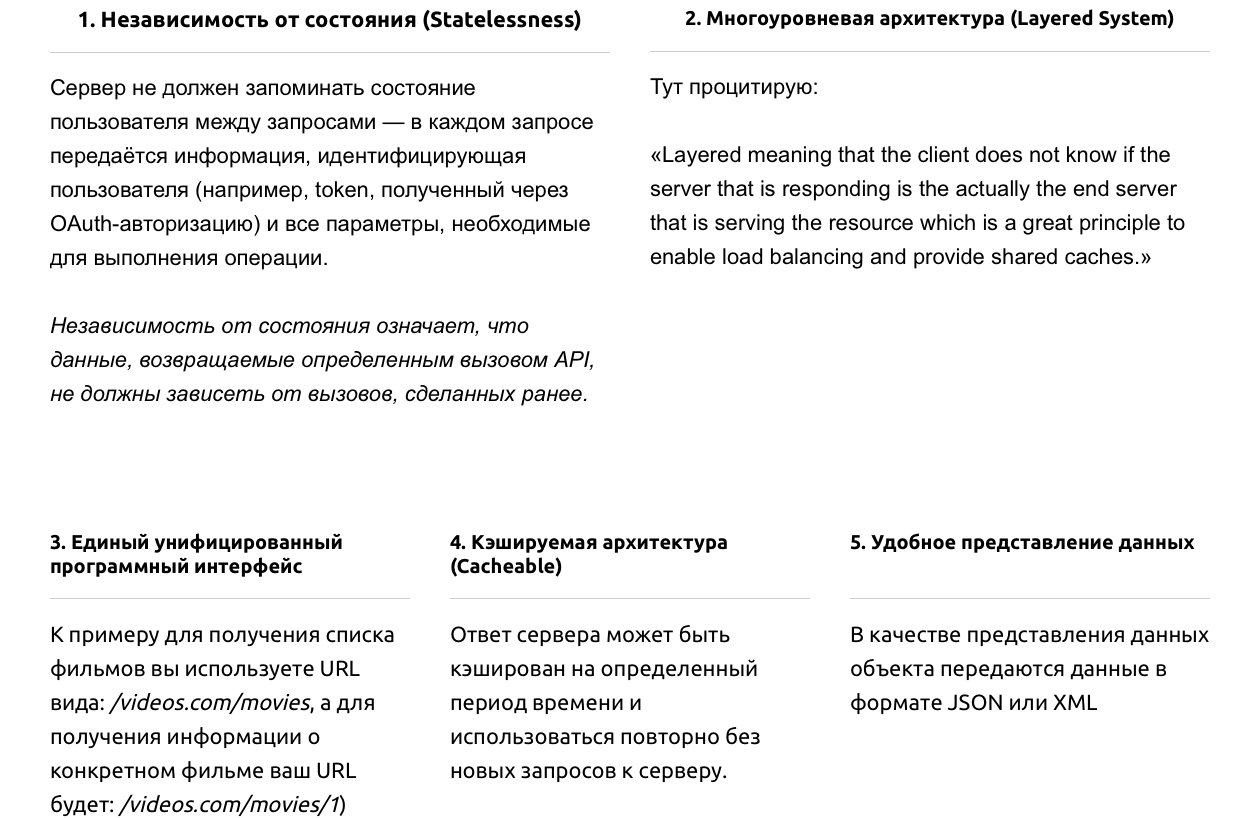
Технология REST не привлекла большого внимания в 2000 году, когда Рой Филдинг (Roy Fielding) впервые представил

В 2004 году W3C выпустил определение ещё одного стандарта под названием RESTful.

За последние несколько лет REST стала преобладающей моделью проектирования Web-сервисов. Фактически REST оказала настолько большое влияние на Web, что практически вытеснила дизайн интерфейса, основанный на SOAP и WSDL, из-за значительного более простого стиля проектирования.

В последние годы этот стандарт стал довольно популярным. На данный момент он используется многими известными сайтами по всему миру, всеми публичными облачными системами.

## **37. Принципы Архитектуры RESTful. Явное использование HTTP-методов REST сервисами. Web сервис с сохранением и без сохранения состояния. Структура URI REST сервиса. Форматы данных используемые web-сервисами REST. Сравнение REST и SOAp/xml Web-сервисов.**



**4 основных типа HTTP запросов REST-архитектуры**

GET — для получения (чтение)

POST — для создания

PUT — для изменения

DELETE — для удаления

JSON, XML, XHTML

**Структура URI аналогичная структуре каталогов.**

Структура URI должна быть простой, предсказуемой и понятной. Один из способов достичь такого уровня удобства использования – построение URI по аналогии со структурой каталогов. Такого рода URI являются иерархическими, исходящим из одного корневого пути, ветвления которого отображают основные функции сервиса.

Например, в сервисе обсуждений различных тем:

* + [http://www.myservice.org/discussion/topics/{topic}](http://www.myservice.org/discussion/topics/%7btopic%7d)
  + [http://www.myservice.org/discussion/{year}/{day}/{month}/{topic}](http://www.myservice.org/discussion/%7byear%7d/%7bday%7d/%7bmonth%7d/%7btopic%7d)

Дополнительные рекомендации, при обдумывании структуры URI для Web-сервисов RESTful:

* Скрывайте расширения файлов серверных сценариев (.jsp, .php, .asp), чтобы можно было выполнить портирование приложений на другую технологию без изменения URI.
* Используйте только строчные буквы.
* Заменяйте пробелы дефисами или знаками подчеркивания (чем-то одним).
* Старайтесь максимально избегать использования строк запросов.
* Вместо использования кода 404 Not Found для URI, указывающих неполный путь, всегда предоставляйте в качестве ответа ресурс или страницу по умолчанию.

**Сравнение REST и SOAp/xml Web-сервисов**

1. RESTful архитектура стала популярна благодаря своей простоте.
2. SOAP поддерживает 16 операций, RESTful всего 4.
3. В случае RESTful приложение должно предоставить все параметры запроса с помощью одной операции, а в случае SOAP число передаваемых параметров за одну операцию ограничено, поэтому для передачи всех параметров требуется выполнение нескольких операций.
4. Вызов SOAP может порождать синтаксические ошибки, обнаруживаемые во время компиляции, в то время как в случае REST проверка вызова откладывается на время исполнения

## **38. Способы реализации ПО промежуточного слоя. Оболочки (Wrappers) или адаптеры (Adapters).**

**Организация промежуточного ПО (midleware)**

При организации ПО промежуточного уровня в системах, основанных на объектных компонентах, часто используются два шаблона проектирования:

* + Оболочки (Wrappers).
  + Перехватчики (Interceptors).

Их применение направлено на достижение открытости системы.

**Wrappers или Adapters**

При создании распределенных проблем на основе уже существующих компонент мы сразу же сталкиваемся с фундаментальной проблемой: интерфейсы, предлагаемые унаследованными компонентами, не поддерживаются всеми компонентами.

**Оболочка или адаптер** – это специальные компоненты, которые обеспечивают приемлемый интерфейс для клиента приложения. Функциями адаптера является преобразование интерфейса компонента-сервер в вид удобный для компонента –клиента.

**Wrapper** реализуется как компонент посредник, обеспечивающий приложению возможность вызова удаленного объекта.

## **39. Перехватчики обращений (interceptors). Обращение к репликам объекта. Перехватчики обращений (interceptors)**

Концептуально перехватчик обращений, прерывает нормальный процесс вызова компонент и позволят исполнить другой фрагмент кода, исходя из нужд приложения.

Базовая идея проста (не перехватываемый вызов): объект А вызывает метод, принадлежащий объекту В, но объект В располагается на другой машине. Такой удаленный вызов метода выполняется за 3 шага:

1.Объект А предлагает, тот же интерфейс, что и объект В. Вызов метода описан в интерфейсе.

2.Вызов метода объекта В преобразуется к нужному виду промежуточным ПО находящемся на машине А.

3.И наконец, вызов объекта преобразуется в сообщение, посылаемое через сетевой интерфейс, как это определено локальной ОС А

**Обращение к репликам объекта B**

* Представим, что объект В имеет несколько реплик. В этом случае мы должны обратиться к каждой реплике. В этом может помочь перехватчик – запроса.
* Хотя объект А ничего не знает о других экземплярах В, но о них знает ПО промежуточного слоя объекта В, выполняющее роль перехватчика обращений уровня запроса.
* В конце концов вызов удаленного объекта должен быть передан по сети, для этого необходимо обратиться к интерфейсу локальной ОС А.
* На этом уровне используется перехватчик уровня сообщения для передачи вызова удаленному объекту на машине В.

## **40. Брокер сообщений. Адаптируемое ПО промежуточного слоя. Замена компонент во время исполнения.**

**Брокер сообщений**

* Уменьшить число адаптеров можно создав промежуточное ПО, которое обеспечит централизованное управление доступом между различными приложениями.
* Часто роль такого ПО выполняет брокер сообщений.
* Приложения просто посылают брокеру запросы, содержащие всю необходимую информацию для доступа к другому приложению.
* В свою очередь брокер, зная как подключиться к требуемому приложению, обращается к нему и получив ответ переправляет его к приложению, инициировавшему запрос.

**Адаптируемое ПО промежуточного слоя**

* Необходимость в адаптируемом промежуточном ПО возникает из-за того, что в окружении РС постоянно возникают изменения: отказ аппаратных средств; сбои электропитания; не соответствие предоставляемого сетью качества обслуживания, требуемому; перенос приложение в другое окружение и т.п.
* Вместо того, чтобы обрабатывать все эти изменения окружающей среды в приложении, эту обработку переносят на уровень промежуточного ПО.
* Такое сильное влияние окружения на распределенную систему требует от разработчиков разрабатывать промежуточного ПО устойчивое к этим изменениям.
* Такое ПО называют адаптивным, т.е. способным к изменению при изменении внешних условий, за счет целенаправленной его модификации, часто без перезагрузки.

**Замена компонент во время исполнения**

Примером такого подхода является **замена программных компонент** в момент их исполнения.

Система может быть сконфигурирована **статически на этапе проектирования** либо **динамически во время исполнения кода**. В последнем случае требуется поддержка позднего связывания не только в языках программирования, но и со стороны операционной системы, когда выполняется загрузка или выгрузка модулей.

## **41. Системная Архитектура. Централизованная и децентрализованная организация РИС. Варианты архитектуры клиент-сервер. Физически Многозвенные системы клиент-сервер. Поведение сервера как клиента в сложных приложениях. Пример работы сервера приложений в WEB приложении**

**Системная Архитектура**

Роль, выполняемая сущностью, является фундаментом для определения архитектуры системы.

Имеется два вида архитектурных систем, определенных на основе ролей, выполняемых элементами РС:

* + Централизованная архитектура (клиент-сервер);
  + Дентрализованная архитектура (peer-to-peer).

**Варианты архитектуры клиент-сервер**

Простейшая организация предполагает наличие всего двух типов машин.

Клиентские машины, на которых имеются программы, реализующие только пользовательский интерфейс или его часть. Серверы, реализующие все остальное, то есть уровни обработки и данных.

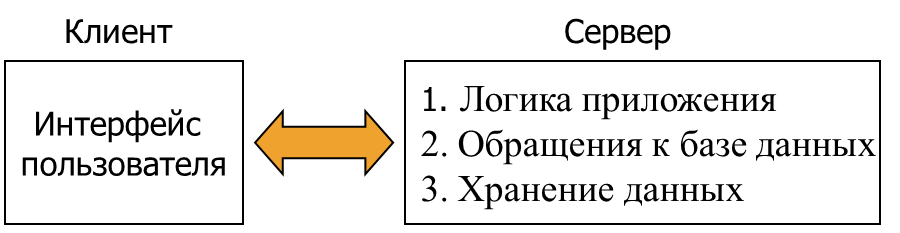
На самом деле такая система не является распределенной: все происходит на сервере, а клиент представляет собой не что иное, как простой терминал.

**Разновидности модели клиент-сервер**

**Толстый клиент.** Такая модель подразумевает объединение в клиентском приложении интерфейса пользователя и обработки данных. Серверная часть в этом случае представляет собой сервер баз данных.



**Тонкий клиент.**  В этом случае клиентское приложение обеспечивает интерфейс с пользователем, а сервер объединяет модули хранения и обработки (толстый сервер).

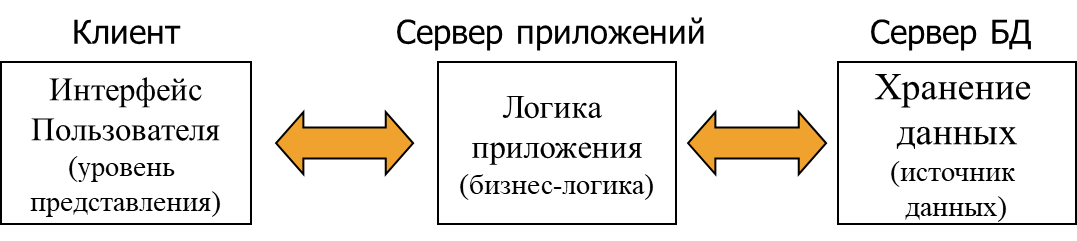


**Многоуровневые системы клиент-сервер.** В этих системах некоторая часть функций, связанных с обработкой данных, либо с доступом к модулям хранения, либо с обеспечением многопользовательского доступа, выделяется в отдельный модуль (*mieddlware*), называемый ПО

**Физически Многозвенные системы клиент-сервер**

**Многозвенная архитектура клиент-сервер** позволяет более разумно распределить модули обработки данных, которые в этом случае выполняются на одном или несколько серверах. Эти программные модули выполняют функции сервера для интерфейса с пользователем и клиента - для серверов баз данных. Многозвенная архитектура клиент-сервер позволяет

* + более точно назначить полномочия пользователей, т. к. они получат права доступа не к самой базе данных, а к определенным функциям сервера приложения.
  + Это повышает защищенность системы (по сравнению с обычной архитектурой) не только от умышленного нападения, но и от ошибочных действий персонала.



**Поведение сервера как клиента в сложных приложениях**

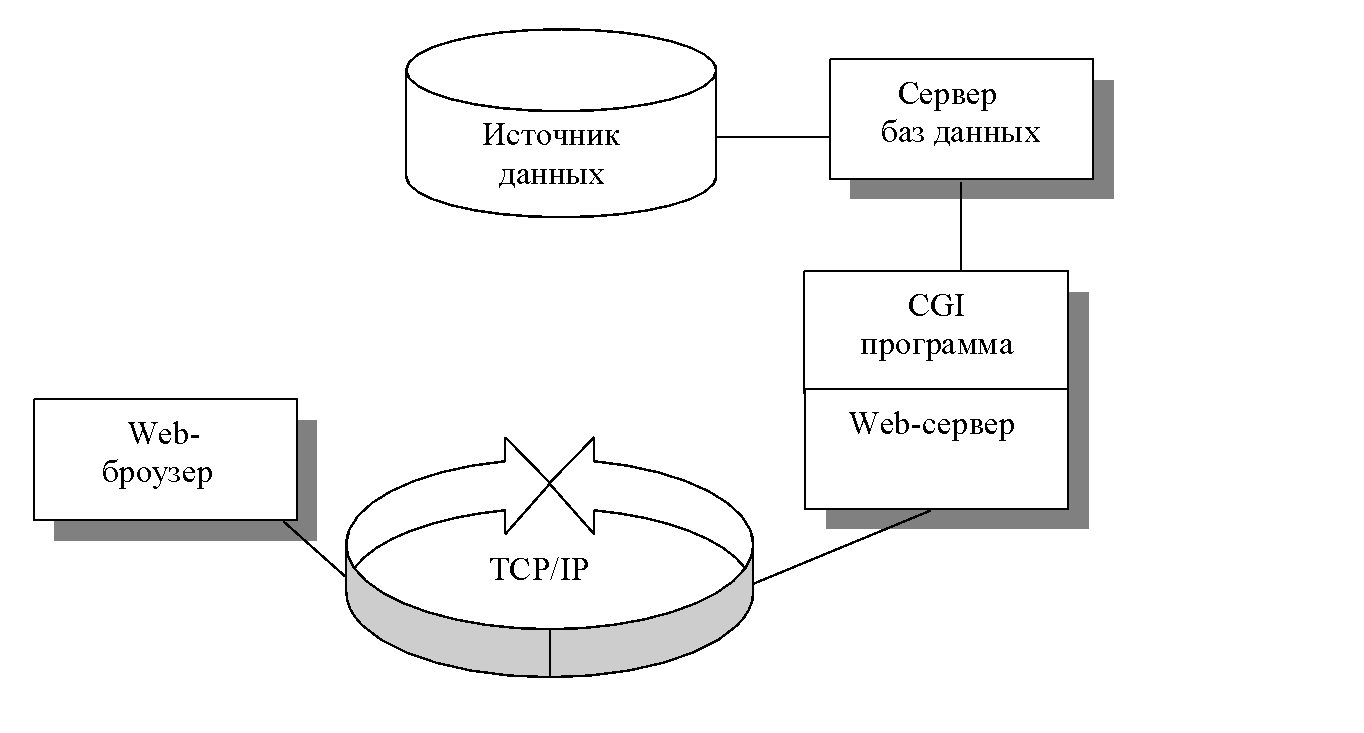
В классической модели клиент-сервер не предполагается поведение машины сервера, подобно машине клиента. Однако такая необходимость иногда возникает. В этом случае мы приходим к **трехзвенной** физической архитектуре клиент-сервер.

В этой архитектуре, программы уровня обработки данных исполняются на отдельном физическом сервере. Типичным примером такой архитектуры являются:

* + системы обработки транзакций, в которых отдельный сервер, называемый монитором транзакций, координирует выполнение отдельных транзакций выполняемых на различных серверах данных.
  + системы с сервером приложений.

**Пример работы сервера приложений в WEB приложении**

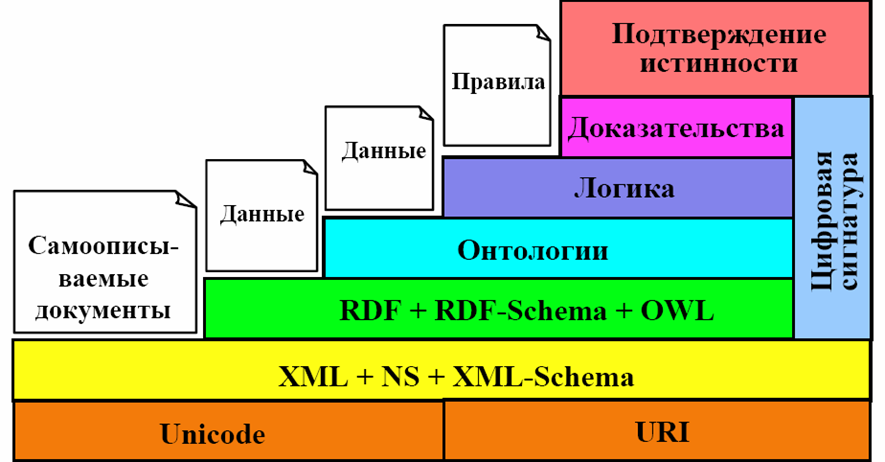
Такая архитектура Web приложения, также является В этом физической трехзвенной (**three-tiered**)



## **42. Понятие семантического Web. Многоуровневое представление Семантического Web по T.Berners-Lee.**

**Семантический Web -** это расширение существующего Web, когда публикуется не только информация в понятном для человека представлении, но и в виде понятном программам, которые способны ее обрабатывать на семантическом уровне.

Многоуровневое представление Семантического Web по T.Berners-Lee



## **43. Средства описания предметной области семантическим Web. Краткая характеристика языков RDF/RDFS/OWL их достоинства и недостатки.**

RDF - язык для описания ресурсов способом, “понятным” компьютеру на семантическом уровне. Свойства также являются ресурсами и имеют URI; Имена свойств берутся из пространств имен, которые указываются в виде префикса.

Недостатки RDFS

* Отсутствуют ограничения на области определений терминов и области значений их свойств
* Отсутствуют ограничения существования/кардинальности
* Отсутствуют способы задания отношений: транзитивности; симметричности; инверсности.
* Трудно обеспечить поддержку рассуждений

Язык онтологий для Семантического Web - Язык OWL (Web Ontology Language)

Три уровня OWL

* OWL full-объединение OWL-синтаксиса и RDF
* OWL DL-ограничен фрагментом логики первого порядка (¼DAML+OIL)
* OWL Lite-“простое для реализации” подмножество OWL DL

Модель OWL

OWL имеет объектно- ориентированную модель, основанную на понятиях:

* Объекты/Экземпляры/Индивидуумы:

–элементы области

–в логике первого порядка -эквивалентны константам

* Типы/Классы/Понятия

–множества объектов, имеющих общие характеристики

–в логике первого порядка –эквивалентны унарным предикатам

* Отношения/Свойства/Роли

–множества пар (троек) объектов

–в логике первого порядка –эквивалентны бинарным предикатам

## **44. Микросервисная архитектура.**

Микросервисы противопоставляются традиционной монолитной архитектуре. Монолит означает, что компоненты продукта взаимосвязаны и взаимозависимы. Если перестает работать один - все остальные тоже «отваливаются».

Предшественником микросервисной архитектуры является [сервис-ориентированная архитектура (SOA)](https://en.wikipedia.org/wiki/Service-oriented_architecture), которая также разделяет бизнес-логику на компоненты. По сути, микросервисная архитектура - частный случай SOA c набором более строгих правил.

У микросервисов есть особые свойства, они же преимущества:

* **Гетерогенность**: возможность построить систему с помощью разных языков программирования и технологий;
* **Децентрализованное управление данными**: каждый микросервис содержит свой набор данных, доступный другим микросервисам только через соответствующий интерфейс;
* **Независимость инфраструктуры**: каждый микросервис - независимая единица, поэтому вносить изменения и разворачивать его можно независимо от других;
* **Масштабируемость**: чтобы увеличить производительность системы, нужно расширить только те сервисы, которые в этом нуждаются.
* **Множество языков, множество возможностей**
* **Использование открытых стандартов типа HTTP**
* **Автоматизация инфраструктуры**
* **Проектирование под отказ (с учетом отказов отдельных микросервисов)**
* **Синхронные вызовы считаются опасными**
* **Каждый сервис работает в отдельном процессе**

**Микросервисы против монолита**

Проще изменить один из микросервисов и сразу внедрить его, чем изменять весь монолит и перезапускать инфраструктуру целиком;

Новые разработчики легче включаются в работу - для этого им не нужно изучать систему целиком, можно работать только над своей частью;

Микросервисы не зависят от какой-либо платформы, поэтому внедрять новые технологии проще, чем в монолит.

**Недостатки:**

* Сложность начальной разработки и создания инфраструктуры. Распределенные системы сложнее разрабатывать, т.к. нужно предусмотреть независимость одного микросервиса от сбоя в другом компоненте;
* Нужно правильно выбрать протоколы общения между компонентами, чтобы взаимодействие было максимально эффективно;
* Для распределенной системы сложно поддерживать строгую согласованность:
  + общие части системы нужно либо складывать в общую библиотеку, но тогда при изменении этой библиотеки нужно будет перезапускать и все зависимые микросервисы,
  + либо хранить общий код в каждом из микросервисов, что сложнее поддерживать

## **45. Архитектуры, основанные на публикации и подписке (п/п). Способы координации в распределенной ИС. Варианты архитектур публикация/подписка. Основная проблема систем публикации/подписки.**

По мере роста размеров РС стало важным иметь архитектуру, в которой зависимость между процессами стала бы как можно меньше.

В качестве таковой была предложена архитектура в которой было введено строгое разграничение между процессами передачи и обработки сообщений и координацией этих процессов.

Идея состояла в том, чтобы взглянуть на РС как на совокупность процессов обработки информации, взаимодействующих между собой.

В этой модели координация заключается в коммуникации и координации между процессами. Координация играет роль клея, связывающего активность процессов в единое целое.

**Способы координации**

В зависимости то вида координации, используемой в системе, можно определить несколько моделей архитектуры подписка/публикация.

В качестве способов координации используются:

* Временная координация;
* Ссылочная (адресная) координация.

**Варианты архитектур публикация/подписка**

В зависимости от комбинации двух факторов координации различают следующие модели архитектуры публикация/подписка:

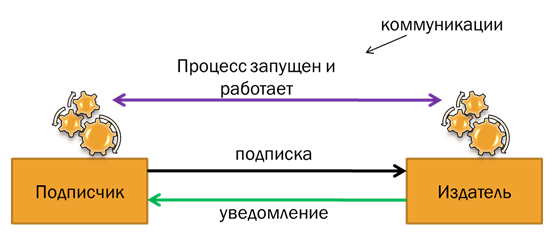
* архитектура П/П с прямой координацией;
* архитектура П/П с координацией через почтовый ящик;
* архитектура П/П с координацией на основе событий;
* архитектура П/П с координацией на основе разделяемых данных.



## **46. Архитектура систем п/п с прямой координацией.**

В этой модели одновременно используются оба вида координации:

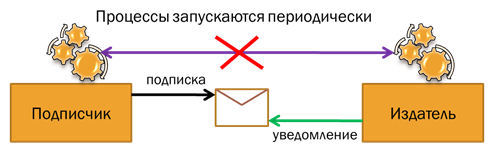
* Координация по времени существует, когда оба процесса запущены и работают).
* Координация по ссылке (адресная) существует, когда имеется явная ссылка на процесс с которым требуется взаимодействие, либо в виде имени либо в виде его идентификатора.

****

## **47. Архитектура систем п/п с координацией через почтовый ящик.**

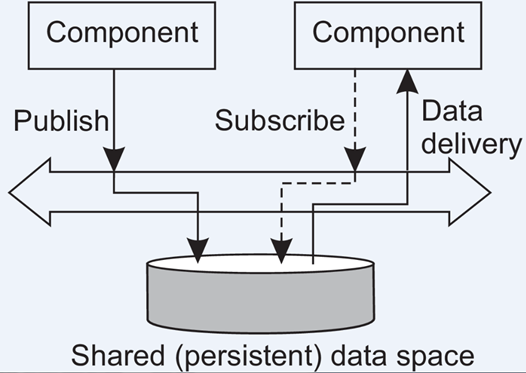
В этой модели:

* Отсутствует координация по времени, оба процесса запускаются и работают не синхронизируясь друг с другом.
  + Подписчик периодически проверяет наличие сообщений в своем почтовом ящике.
  + Отправитель (издатель) периодически активизируется для отправки сообщения в почтовый ящик другого процесса.
* Координация по ссылке (адресная) существует, в виде адреса почтового ящика.

****

## **48. Архитектура систем п/п на основе общего пространства данных.**

В случае отсутствия обоих видов координации получается модель с общей шиной данных.

****

Пример: Linda - программная модель, разработанная в 1980 году. Разделяемое пространство данных в модели Linda называется пространство записей (кортежей). Это пространство поддерживает три операции:

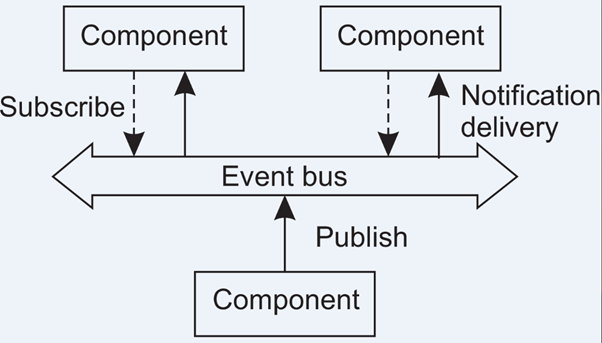
* in(t): извлечь (с удалением) запись, соответствующую шаблону t;
* out(t): занести запись по шаблону t.

Операции in(t) и out(t) взаимно блокируют друг друга.

* zd (t): получить копию записи по шаблону **t;**

## **49. Архитектура систем п/п на основе событий. Варианты Архитектуры п/п на основе событий. Принцип обмена данными между издателем и подписчиком в системах с шиной событий**

В этой модели ссылочная координация отсутствует, но поддерживается во времени.



Для получения уведомления процесс подписчик должен всегда находиться в рабочем состоянии. Отсутствие ссылочной информации не позволяет процессам иметь явные знания друг о друге. Такая архитектура соответствует коммуникациям через общую шину сообщений.

**Варианты Архитектуры п/п на основе событий**

В зависимости от способа описания события различают два варианта систем с архитектурой публикации/подписке на основе событий:

* системы публикации/подписки основанные на теме;
* системы публикации/подписки основанные на содержимом (контенте)

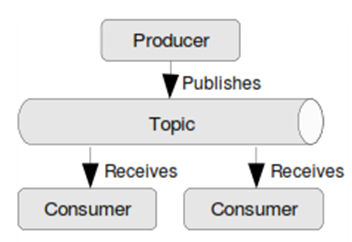
**система публикации/подписки основанная на теме**

Событие описывается набором атрибутов – списком пар (атрибут, значение).

Подписка должна быть направлена к промежуточному ПО с описанием события (список пар атрибутов и их значений), в котором заинтересован подписчик.

Уведомление описывает опубликованное событие, когда оно становится доступным другим процессам для чтения. Оно также должно содержать список пар атрибутов и их значений, характеризующих событие.

Получив уведомление подписчик определяет соответствует ли событие подписке.



**Принцип обмена данными между издателем и подписчиком в системах с шиной событий**

Условием обмена данными является совпадение подписки на событие и уведомление о событии.

Во многих случаях событие на самом деле соответствует данным, ставшим доступными. В этом случае при совпадении имеется два сценария:

* Программное обеспечение промежуточного уровня (ПУ) может решить направить подписчикам уведомления вместе с ассоциированными с этим событием данными. Это называется **процессом с совпадением подписки**.
* Программное **ПУ передает только уведомление**. При этом подписчики смогут самостоятельно считать требуемые данные. ПО ПУ не предоставляет услуг по хранению данных. Услуги хранения оказываются отдельным сервисом.

**Основная проблема систем публикации/подписки**

* События могут легко запутать работу подписчиков. В качестве примера рассмотрим такую подписку: "Уведомить, когда номер 1060 будет освобожден и дверь в номер будет не заперта".
* Обычно распределенная система, поддерживающая такие подписки, может быть реализована путем установки независимых сенсоров (датчиков) для мониторинга занятости номера (например, датчики движения) и регистрации состояния дверного замка.
* Для обеспечения надежной работы такой системы, необходимо сформировать (скомпоновать) такие примитивные события, помещаемые в публикуемые данные, на которые можно было бы подписать процессы получатели данных.
* Компоновка событий оказывается сложной задачей, особенно когда простейшие (примитивные) события генерируются источниками, распределенными по всей системе.
* Очевидно, что в системах публикации/подписки подобным этой, основная проблема кроется в эффективности и масштабируемости реализации сравнения подписок и уведомлений.

# **Тема 7. Децентрализованная организация РС: системы peer-to-peer (одноранговые системы)**

## **50. Сравнение систем P2P и клиент-сервер. Сравнение принципов построения систем P2P и клиент-сервер. Способы реализации распределенности в системе. Задачи решаемые с помощью р2р систем. Принципы построения РС Р2Р.**

В отличие от классической клиент-серверной архитектуры в архитектуре Р2Р каждый узел, входящий в РС, может играть роль и клиента, и сервера. Предоставляя свои либо используя чужие ресурсы.

Можно выделить следующие проблемы архитектуры клиент-сервер:

* Проблемы масштабируемости.
* Проблема высокой связности узлов.

Р2Р системы обладают следующими преимуществами:

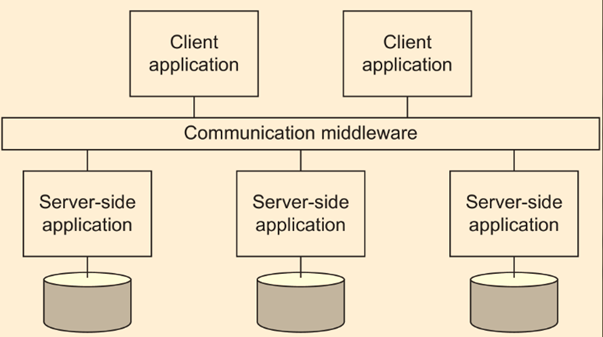
* Слабая зависимость от централизованных сервисов и ресурсов.
* Система допускает сильные изменения в структуре, сохраняя при этом свою работоспособность.
* Высокая масштабируемость

Четкой границы между централизованной и децентрализованной архитектурами не существует.

**Различают два способа реализации распределенности:**

**Вертикальная распределенность.** Реализуется путем разбиения приложения на логические уровни, связанные иерархически. Такая организация характерна для многозвенных клиент-серверных архитектур.

Каждый логический уровень размещается на отдельном узле РС



**Горизонтальная распределенность.** В этом случае клиент или сервер могут одновременно исполняться на одном и том же узле, разделяя между собой его ресурсы.Системы, поддерживающие горизонтальную распределенность получили название пиринговых систем

**Задачи**

* **Уменьшение/распределение затрат.** Серверы централизованных систем, которые обслуживают большое количество клиентов, обычно несут на себе основной объем затрат ресурсов (денежных, вычислительных и др.) на поддержание вычислительной системы. P2P архитектура может помочь распределить эти затраты между узлами сети. Так как узлы, как правило, автономны, важно, чтобы затраты были распределены справедливо.
* **Объединение ресурсов.** Каждый узел в P2P-системе обладает определенными ресурсами (вычислительные мощности, объем памяти). Приложения, которым необходимо большое количество ресурсов используют возможность объединения ресурсов всей сети для решения своей задачи. При этом важны как объем дискового пространства для хранения данных, так и пропускная способность сети.
* **Повышенная масштабируемость.** Масштабируемость определяет количество систем, которые могут быть достигнуты из одного узла, сколько систем могут функционировать одновременно, сколько пользователей может пользоваться сетью, сколько памяти может быть использовано.

**Принципы построения**

В распределенной системе Р2Р каждый узел участник знает некоторое число логических соседей, с которыми он может поддерживать обмен напрямую, без посредников, посылая и получая в ответ сообщения по сети.

Этот набор соседей формирует логический граф связности для всех узлов системы. Это граф часто называют **наложенной сетью Р2Р**.

Системы Р2Р должны соответствовать следующим критериям:

* **Самооганизация:** система самостоятельно адаптируется к подключению или отключению узла к системе. Пиры используют локальную информацию, получаемую от своих соседей для организации взаимодействия друг с другом.
* **Распределенность:** отсутствует централизованное управление поведением узла в системе.
* **Масштабируемость:** система может масштабироваться неограниченно, избегая таким образом проблем «бутылочного горла», отказов отдельных узлов и проблем перегрузки узлов.

**Системы Р2Р можно классифицировать по двум признакам:**

* По степени централизации распределенной системы.
* По наличию или отсутствию структуры распределенной системы.

## **51. Классификация P2P по степени централизации.**

**Централизованные**

В этом случае некоторая группа узлов отвечает за выполнение критически важных для всей системы операций, например:

* Аутентификация пиров в системе.
* Определение расположения узлов и ресурсов, принадлежащих узлам.

Эти управляющие узлы могут располагаться в одной LAN, но могут быть разбросаны географически, но в любом случае между ними должны существовать широкополосные каналы связи.

**недостатки**

* Управляющие сервера являются потенциальными точками отказа.
* Такое решение является плохо масштабируемым.

Эти недостатки отсутствуют в частично централизованных (гибридных) системах (рассматриваются в разделе гибридные архитектуры Р2Р систем).

**Децентрализованные**

При очень большом числе пиров гибридные системы не способны обеспечить корректную обработку всех запросов. Они перегружены запросами и не способны их обработать за приемлемое время.

В децентрализованных системах каждый пир, принадлежащий к наложенной сети, обеспечивает обзор ресурсов только у своих логических соседей.

Для обеспечения устойчивости к отказам отдельных узлов информация о ресурсах, принадлежащих каждому пиру, реплицируется всем его соседям по наложенной сети.

Децентрализованная система не имеет точек единственного отказа, однако она менее эффективна чем соответствующая централизованная система.

## **52. Гибридные архитектуры р2р систем. Системы с граничными серверами. Архитектура bittorrent.**

Существует множество архитектур РС в которых успешно сочетаются архитектуры клиент сервер и децентрализованные архитектуры Р2Р систем.

В этих системах все узлы делятся на несколько типов в зависимости от иерархической организации. В двухуровневой системе имеется два вида узлов:

* суперузлы; мощные узлы, связанные между собой хорошими каналами связи. Хранят метаданные о расположении ресурсов и
* рядовые пиры; связаны с одним или несколькими суперузлами.

**с граничными серверами**

Эти системы разворачиваются в Интернет, при этом на границах сетей сервис провайдеров размещаются сервера этих систем.

Конечные пользователи этих систем располагаются в сетях сервис провайдеров и подключаются к распределенной системе через граничные сервера.

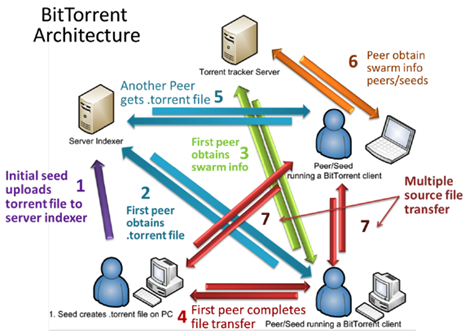
Основное назначение граничных серверов предоставлять контент для пользователей, возможно после предварительной фильтрации и перекодировки.

Базовая модель состоит в том, что для организации граничный сервер является источником всего контента, получаемого из Интернет. Этот сервер может использовать другие граничные сервера, например для репликации Web страниц.

**архитектура BITTORRENT**

Если узел хочет опубликовать файл или набор файлов, то программа-клиент BitTorrent сети разделяет передаваемые файлы на части и создает файл метаданных (идентификатор раздачи), который содержит следующую информацию:

* URL трекера (супер-сервера, который хранит актуальную информацию об активных узлах которые имеют отдельные части файла);
* Общая информация о файлах (имя, длина и пр.);
* Хеш-суммы SHA1 сегментов раздаваемых файлов;
* Passkey пользователя – ключ, который однозначно определяет пользователя, загрузившего файл;
* Хеш-суммы файлов целиком (не обязательно);
* Альтернативные источники – адреса альтернативных трекеров, на которых можно найти информацию по данному файлу (не обязательно).



## **53. P2P системы на основе технологии распределенного реестра. Понятие распределенного реестра (DLT). Виды распределенных реестров. Классификация распределенных баз данных. Где используются DLT ?**

**Распределенный реестр** (DL) или распределенная база данных записи которой хранятся одновременно на всех узлах сети Р2Р. Распределенный реестр (DL) имеет несколько ключевых особенностей:

* отсутствие центрального администратора и центральной (генеральной) копии б/д;
* совместное использование данных всеми узлами с синхронизацией данных по заданному алгоритму;
* децентрализованное географическое распределение копий базы данных между всеми узлами р2р сети.

Каждый узел вносит изменения в реестр независимо от других узлов, затем все они голосуют за внесение изменений и при достижении консенсуса реестр дополняется новыми данными. Каждый участник сети при этом обладает собственной идентичной копией реестра, а сами изменения добавляются в течение нескольких минут.

В отличие от традиционных баз данных, распределительные реестры намного лучше защищены от атак и несанкционированного изменения данных, поскольку копии географически отдалены друг от друга и для хакерского изменения потребуется произвести атаку сразу на все узлы связи. Кроме того, ноды обнаружат несанкционированные изменения и не поддержат внесение данных.

**Условно все виды распределенных реестров можно разделить на:**

* **Публичные.** Применяются в большинстве криптовалют и представляют собой базу данных с открытым исходным кодом. Они работают на алгоритмах подтверждения выполнение некоторого объема вычислений. В такой системе каждый участник может загрузить себе на локальное устройство базу данных и участвовать в согласованном процессе внесения изменений. Также любой желающий может просмотреть всю добавленную информацию.
* **Федеративные.** Федеративные базы данных работают под управлением группы людей. В отличие от открытых реестров они не поддерживают внесение новых данных всеми желающими. Процесс изменения реестра контролируется исключительно заранее выбранными узлами сети. Применяются они преимущественно в банковском секторе и обеспечивают большую конфиденциальность.
* **Частные.** Право на внесение изменений в такой реестр имеет только определенная централизованная организация. Информация может быть открыта для публичного чтения или быть ограниченной в произвольной степени. Частные распределительные реестры, как правило, используются компаниями для хранения внутренней информации и проведения аудита. Такие системы более уязвимы, нежели публичный блокчейн, но позволяет модернизировать устаревшие системы хранения информации в компаниях.

Основной областью использования распределительных реестров сейчас остаются криптовалюты, где по такой системе хранится информация о транзакциях и производится подсчет баланса кошельков пользователей.

Полностью понимая технологию правительство и частный сектор смогут существенно модернизировать процесс хранения и использования информации. Например, правительство многих стран постепенно внедряет технологию DLT в разные секторы своей работы.

## **54. Классификация Р2Р систем по их Структуре. Структурированные Р2Р системы. Индекс ресурса структурированной P2P системы. Распределенная хэш таблица(DHT). Поиск узла в структурированной P2P системе. Поиск с помощью маршрутизации документов.**

Структура Р2р системы определяется структурой наложенной сети, которая может иметь одну из известных топологий:

* кольцо;
* Двоичное дерево;
* Решетка; И т.п.

Эта топология используется для эффективного поиска данных (ресурсов).

Характеристикой структурированной Р2Р системы является так называемый индекс, который однозначно определяет каждый экземпляр данных, поддерживаемый в системе и соответственно каждый узел, располагающий каким-либо ресурсом. Соседство в системе определяется по близости значений индексов у узлов. В качестве такого индекса используется ключ хэш функции.

**Распределенная хэш-таблица**

Каждому узлу структурированной P2P системы назначается идентификатор из одного того же набора значений хэша. И каждый узел становится ответственным за хранение данных, связанных с определенным набором ключей. По сути, система описывается с помощью таблицы хэшей - распределений хэш таблицы (DHT).

**Поиск узла**

Для того чтобы найти узел, ассоциируемый с ключом, необходимо выполнить операцию поиска узла по ключу: node = lookup (key).

В задаче поиска топология наложенной сети играет ключевую роль.

Любой узел однозначно связан с некоторым ключом, что позволяет организовать эффективную маршрутизацию для поиска узла, ответственного за хранение ресурса.

**Поиск с помощью маршрутизации документов**

Данный метод обеспечивает поиск документов без участия центрального индекса, средствами самой вычислительной сети.

В основе данного метода лежит принцип присвоения уникальных идентификаторов каждому узлу вычислительной сети, а также каждому ресурсу (документу, сервису и др.), который данная сеть может предоставлять.

Когда какой-либо пир производит публикацию ресурса в сети, каким-либо образом вычисляется идентификатор данного ресурса. При этом, идентификаторы узлов сети и ресурсов имеют единую область возможных значений.

Далее, копия данного ресурса (или ссылка на него) связывается с узлом, который имеет наиболее похожий идентификатор.

Данная процедура производится следующим образом:

* Производится сравнение идентификатора ресурса с идентификаторами всех соседних узлов.
* Если идентификатор текущего узла ближе всего (по некоторой метрике) к идентификатору документа, то процесс трассировки завершается.
* Если один из идентификаторов соседних узлов ближе к идентификатору документа, чем идентификатор текущего узла, то ссылка на данный ресурс копируется на узел с более близким идентификатором и процесс повторяется с п. 1.

Поиск ресурса производится по аналогичному алгоритму, но вместо копирования документа происходит трансляция запроса (в запросе содержится идентификатор запрашиваемого ресурса) от узла, инициировавшего запрос к узлу, идентификатор которого ближе всего соответствует идентификатору запрашиваемого документа.

Соответственно, в процессе трансляции данного запроса должен появиться такой узел, который хранит информацию об интересуемом документе, если данный документ находится в соответствующей части сети.

К недостаткам такого подхода можно отнести необходимость знания точного идентификатора документа, который необходимо найти в сети (невозможность поиска по отдельным атрибутам документа), а также возможность образования «островов», затрудняющих алгоритм поиска.

## **55. Неструктурированные P2P системы. Методы поиска узлов в неструктурированных Р2Р системах. Метод затопления. Метод случайного блуждания. Метод поиска, основанный на политике.**

Они не имеют определенной топологии.

Каждый узел в такой системе поддерживает список соседних узлов, а об остальных он ничего не знает.

В результате наложенная сеть такой системы представляет собой случайный граф: - граф в котором дуга между вершинами <u,v> существует с некоторой вероятностью P[<u,v>]. В идеале эта вероятность имеет одно и тоже значение, для всех пар вершин.

При присоединении узла к Р2Р системе, он контактирует с общеизвестными в системе узлами и получает от них список других пиров, от которых он может узнать о еще большем числе пиров.

В таких системах, если пир хочет найти какой-либо ресурс, то он не может следовать какой-то определенной процедуре маршрутизации. Он должен прибегнуть к специальной процедуре поиска, установленной в этой системе.

**Имеется два предельных метода поиска в таких системах:**

* Метод затопления (flooding).
* Метод случайного блуждания (random walk).

**Затопления**

Узел u посылает запросы ко всем известным ему соседям.

Если узел v имеет требуемые данные, то он либо посылает их напрямую узлу, породившему запрос, либо отсылает их к узлу форвардеру от которого он получил запрос.

Если узел не имеет данных, то он форвардирует запрос ко всем своим соседям.

Если узел уже получал такой запрос, то он не будет повторно на него отвечать.

Для того, чтобы избежать избежать сильного затопления системы, для запросов устанавливается время жизни (TTL). Каждый промежуточный узел обрабатывающий запрос уменьшает TTL на 1. Когда TTL станет равным 0 запрос удаляется из системы.

От выбора значения TTL сильно зависит эффективность поиска и степень затопления системы запросами.

**Случайного блуждения**

Узел u просто опрашивает соседей выбирая их случайным образом. Например, пусть u выберет для опроса узел v.

Если v имеет требуемые данные, то он напрямую ответит u, если нет, то v форвардирует запрос своему соседу, выбранному случайным образом.

Очевидно, что этот метод порождает значительно меньший трафик, но для поиска нужного ресурса может потребоваться значительно больше времени.

Для уменьшения времени ожидания узел источник процедуры поиска может одновременно стартовать n процедур случайного блуждания. В этом случае в этом случае время поиска снижается в n раз.

Случайное блуждание также требует своей принудительной остановки. Для этого можно также использовать TTL, либо можно установить условия, при которых узел, принявший запрос сам, определит нужно форвардировать запрос случайным образом далее или нет.

Для неструктурированных систем также необходимо определить порядок сравнения для определения признака нахождения искомых данных. Это могут быть те же методы, которые применяются в структурированных системах Р2Р.

**Метод поиска основанный на политике**

Этот метод занимает промежуточное положение между методами затопления и случайного блуждания.

В этом методе для выбора узлов в качестве соседей используется некоторая установленная политика.

Например:

* в качестве узлов, к которым перенаправляются запросы выбираются узлы, которые ранее более успешно выполняли поиск, по сравнению с другими узлами.
* в качестве узлов, к которым направляются запросу выбираются узлы, имеющие наибольшее число соседей.
* и т.п.

## **56. Иерархически организованные Р2Р системы. Пример иерархической Р2Р системы – Skype.**

В не структурированных системах по мере их роста поиск данных может стать невозможен. Источник этой проблемы масштабируемости очевиден – отсутствие детерминированой процедуры маршрутизации запросов поиска ресурсов.

В качестве альтернативы во многих неструктурированных системах создаются специальные узлы, хранящие индекс экземпляров данных имеющихся в системе.

Имеются и другие ситуации в которых отказ от симметричной природы р2р систем имеет смысл. Например, сети доставки контента (CDN).

В CDN сетях узлы системы хранят копии Web документов отдельных сайтов Интернет к которым пользователям требуется быстрый доступ. Для поиска узлов используются брокеры, которые хранят сведения о степени используемости ресурсов и доступности узлов, что позволяет выбирать наиболее подходящий из узлов для получения ресурсов.

Узлы, играющие роль брокеров, а также узлы, поддерживающие индекс ресурсов, называются супер-узлами. Супер-узлы часто образуют сеть р2р. Обычные узлы выполняющие роль клиентов супер-узлов называют слабыми.

В этих сетях супер-узлы должны всегда оставаться доступными в сети. Это создает проблемы надежности, которые можно компенсировать, путем развертывания системы копирования восстановления между парами супер-узлов, и обеспечением подключения рядовых узлов к обеим супер-серверам.

В сетях с супер-узлами, возникают и другие проблемы, например как отбирать узлы для роли супер-узлов.

Самой популярной на сегодняшний день службой Интернет-телефонии является Skype, созданная в 2003 году шведом Никласом Зеннстромом и датчанином Янусом Фриисом.

В настоящее время Skype принадлежит корпорации Microsoft, которая приобрела ее за $8,5 млрд. в мае 2011 года.

В состав системы входят следующие элементы:

* Skype-login сервер – единственный централизованный элемент Skype-сети, обеспечивающий авторизацию Skype-клиентов.
* Обычный узел (Skype Client) – обычный конечный узел в сети.
* Супер-узел (Super node) – узлы, играющие роль роутеров в сети Skype.
* Выделенные узлы для установки связи со стационарными телефонными линиями.

Каждый узел Skype-сети хранит перечень IP-адресов и портов известным ему super-узлов в динамически обновляемых кэш-таблицах. Начиная с версии Skype 1.0, кэш-таблицы представляют собой простой XML-файл, в незашифрованном виде записанный на диске в домашней директории пользователя.

Любой узел, имеющий публичный IP-адрес (т.е. тот, который маршрутизируется в Интернет) и обладающий достаточно широким каналом, автоматически становится super-узлом и пропускает через себя трафик обычных узлов, помогая им преодолеть защиты типа брандмауэров или трансляторов сетевых адресов (NAT) и равномерно распределяя нагрузку между хостами.

Единственным централизованным элементом является Skype-login сервер, отвечающий за процедуру авторизации Skype-клиентов и гарантирующий уникальность «позывных» для всей распределенной сети.

## **57. Достоинства и недостатки P2P систем.**

**Можно выделить следующие основные преимущества пиринговых систем:**

* высокая масштабируемость, связанная с равномерным распределением вычислительной нагрузки на всех участников сети;
* стабильность работы сети, обусловленная отсутствием таких «узких мест» как выделенный сервер, обрабатывающий все сетевые запросы;
* возможность объединения ресурсов отдельных участников сети, и их предоставление другим участникам;
* распределение совокупных затрат на предоставление ресурсов между участниками сети.

**Стоит упомянуть о следующих недостатках и особенностях функционирования р2р систем:**

* в одноранговых сетях не может быть обеспечено гарантированное качество обслуживания: - любой узел, предоставляющий те или иные сервисы, может быть отключен от сети в любой момент;
* индивидуальные технические характеристики узла могут не позволить полностью использовать ресурсы P2P сети (каждый из узлов обладает индивидуальными техническими характеристиками что, возможно, будет ограничивать его роль в P2P-сети и не позволят полностью использовать ее ресурсы: - низкий рейтинг в torrent-сетях, или LowID в eDonkey могут значительно ограничить ресурсы сети, доступные пользователю);
* при работе того или иного узла через брандмауэр может быть значительно снижена пропускная способность передачи данных в связи с необходимостью использования специальных механизмов обхода;
* участниками одноранговых сетей в основном являются индивидуальные пользователи, а не организации, в связи с чем возникают вопросы безопасности предоставления ресурсов: владельцы узлов P2P-сети, скорее всего, не знакомы друг с другом лично, предоставление ресурсов происходит без предварительной договоренности;
* при увеличении числа участников P2P сети может возникнуть ситуация значительного возрастания нагрузки на сеть (как с централизованной, так и с децентрализованной структурой);
* в случае применения сети типа P2P приходится направлять значительные усилия на поддержку стабильного уровня ее производительности, резервное копирование данных, антивирусную защиту, защиту от информационного шума и других злонамеренных действий пользователей.

# **Тема 8. Электронная платежная система Bitcoin на базе распределенного реестра blockchain**

## **58. Система платежей bitcoin. История Bitcoin. Электронная монета bitcoin (BTC). Свойства Bitcoin как денег.**

Bitcoin – это криптовалюта и децентрализованная платежная система Р2Р. Эмиссия Bitcoin ограничена — количество монет не превысит 21,000,000. Согласно расчетам, выпуск Bitcoin закончится в 2045 году.

Биткойн функционирует без какого-либо контролирующего органа или центрального банка; обработка транзакций и эмиссия осуществляются коллективно участниками сети.

Биткойн имеет открытый исходный код; его архитектура известна всему миру, никто не владеет и не контролирует Биткойн, но все могут стать участниками сети.

Благодаря своим уникальным свойствам, Биткойн обладает возможностями, которые не предоставляла до этого ни одна платежная система.

В качестве электронной денежной единицы (монеты) определяется последовательность цифровых подписей хэшей отдельных операций с криптовалютой.

По умолчанию, первая транзакция по генерации хэша является специальной, создающей новую монету, которая принадлежит создателю блока. Такая схема поощряет честных участников сети, стимулируя их поддерживать работу сети, а также решает вопрос о начальном распределении денежной массы в отсутствие центрального эмитента

Электронные деньги могут быть переданы от одного владельца к другому. Такой обмен называется транзакцией.

Bitcoin обладает таким свойствами реальных денег, например, золота, как – материальность, сложность добычи, ограниченность, сложность подделки.

**Материальность Bitcoin**

Материальность реальных денег (например, золота). Один слиток золота нельзя дважды обменять на услугу или товар. Т. е. в один момент времени он может быть либо у продавца, либо у покупателя.

В Bitcoin подобное поведение обеспечено механизмом транзакций. Все транзакции объединяются в цепочки. Каждая транзакция берет монетки из одной или нескольких существующих транзакций и указывает, кому они предназначаются. Поэтому всегда можно проверить всю цепочку на валидность.

**Сложность добычи Bitcoin**

Золото нельзя скопировать — его можно только добыть. Но это очень затратный процесс как по времени, так и по ресурсам. Частично из-за этого золото ценится так высоко.

В Bitcoin процесс добычи монеток тоже требует ресурсов и времени. Но в данном случае это не человеческие ресурсы, а компьютерные. На добычу 1 BTC в среднем уходит 10мин.

Условно ограниченный ресурс. Общее количество Bitcoin ограничено числом 21106. Каждый BTC разбивается на более мелкие единицы: 1 сатоши = 10-8 BTC.

Сложность подделки. Гарантируется надежностью, используемых криптографических средств.

## **59. Криптография Bitcoin. криптография на эллиптических кривых. Электронная цифровая подпись. Личный (private) и публичный (public) ключи. Адрес узла. Подпись и проверка сообщений между узлами**

Криптография — это фундаментальная часть Bitcoin. Без нее вообще бы ничего не заработало, поэтому начинать нужно именно отсюда. В рамках bitcoin используются следующие криптографические методы:

* Шифрование на эллиптических кривых.
* Электронная цифровая подпись.

Ассиметричная система ключей шифрования с использование узлами открытых (публичных) и закрытых (личных, частных) ключей основанных на ECC, в частности ECDSA.

Сегодня криптосистемы на эллиптических кривых используются в TLS, PGP и SSH, важнейших технологиях, на которых базируются современный веб и мир ИТ.

**Замечание**. До того, как ECC стала популярной, почти все алгоритмы с открытым ключом основывались на RSA, DSA и DH, альтернативных криптосистемах на основе модулярной арифметики. RSA и компания по-прежнему популярны, и часто используются вместе с ECC.

В Bitcoin криптография используется также для:

* Определения адреса узлов с сети blokchain;
* Подписи и проверки сообщений;
* При представлении данных в нужном формате.
* Для представления ссылок в сети р2р на основе публичных ключей

**Публичный (открытый) ключ.**

Пусть — наш приватный ключ, G, тогда публичный ключ K=G\*k. То есть, фактически, публичный ключ — это некоторая точка, лежащая на кривой SECP256k1.

**Два важных нюанса.**

* Во-первых, несложно видеть, что операция получения публичного ключа определена однозначно, то есть конкретному приватному ключу всегда соответствует один единственный публичный ключ.
* Во-вторых, обратная операция является вычислительно трудной и, в общем случае, получить приватный ключ из публичного можно только полным перебором первого.

Первый и самый распространенный вариант его записи — uncompressed формат, по 32 байта для X и Y координат. Чтобы не возникало путаницы, используется префикс 0x04 и того 65 байт.

Это не самый оптимальный способ хранить публичный ключ. Второй формат называется compressed. Суть его в следующем: публичный ключ — это точка на кривой, то есть пара чисел, удовлетворяющая уравнению

А значит можно записать только Х координату и, если нам понадобится Y координата — просто решаем уравнение. Тем самым мы уменьшаем размер публичного ключа почти на 50%!

Единственный нюанс — если точка лежит на кривой, то для ее Х координаты очевидно существует два решения такого уравнения: одна из точек будет иметь четную Y координату, а вторая — нечетную. В первом случае используется префикс 0x02, во втором — 0x03.

Адрес узла получается из публичного ключа однозначным образом. Более того, провести обратную операцию невозможно, так как используются криптографически стойкие хэш функции — RIPEMD160 и SHA256.

Вот алгоритм перевода публичного ключа в адрес:

* Возьмем приватный ключ
* Получим из него публичный ключ в uncompressed формате
* Считаем RIPEMD160(SHA256(public\_key))
* Переводим результат в Base58Check с префиксом 0x00
* Это и есть адрес.

## **60. Сеть Blockchain. Понятие транзакции и проблемы достоверности транзакций в системе BITCOIN. Входы и выходы транзакции. Структура транзакции. Вычисление хэша транзакции.**

**Транзакция.** Очередной владелец отправляет сумму в BTC следующему, подписывая хэш предыдущей транзакции и публичный ключ будущего владельца своим личным ключем и присоединяя эту информацию к переводимой сумме. Получатель может проверить каждую подпись, чтобы подтвердить корректность всей цепочки владельцев.

Сеть Биткойн представляет собой р2р систему распределенного реестра. В которой в качестве данных хранятся цепочки блоков с их хэшами.

Каждый участник может запустить свой узел с полной копией блокчейна.

Полные ноды, которые могут записывать транзакции в блокчейн, называются **узлами консенсуса** или **майнерами**.

Полные ноды, которые только проверяют правильность транзакций называются узлами **аудита**.

Легкие клиенты не хранят полных копий блокчейна, а взаимодействуют с сетью, используя полные ноды.

Большинство пользователей для совершения транзакций используют именно легких клиентов или web кошельки. Все ноды связаны друг с другом. При таком наборе элементов архитектура сети становится более устойчивой

Операции с криптовалютой (добыча, продажа/ покупка, обмен и т.п.) называются **транзакциями**.

Система работает по следующим правилам:

1) Каждый узел пытается подобрать хэш для нового блока, удовлетворяющий текущей сложности, в случае успеха узел получает комиссию (операция mining-добыча). Комиссионные выплачиваются узлам и за операции проведения и проверки правильности транзакций.

2) Как только такой хэш найден, создается блок, содержащий транзакцию добычи N монет, информация о нем отправляется другим узлам сети.

3) Новые транзакции об операциях с криптовалютой включаются в соответствующий блок и рассылаются всем узлам.

4) Узлы принимают блок, только если все транзакции в нем корректны и не используют уже потраченные средства.

5) Свое согласие с новыми данными узлы выражают, начиная работу над следующим блоком и используя хэш предыдущего блока в качестве новых исходных данных.

**Проблемы транзакций:**

* 1. **Двойной платеж.** Проблема, транзакции состоит в том, что получатель не может определить, сколько раз бывший владелец потратил данную монету. Традиционное решение заключается в проверке центральным доверенным лицом каждой транзакции. Недостаток этого подхода в том, что от компании-эмитента зависит судьба всей денежной системы, так как она подобно банку контролирует каждую проходящую через нее транзакцию.
  2. **Порядок следования транзакций**: чтобы избавить схему от посредника, участникам необходимо открыто публиковать транзакции, а также уметь приходить к согласию относительно единого порядка их следования. Получателю нужно доказательство того, что для каждой транзакции из цепочки большинство пользователей согласны считать ее первой.
  3. **Подделка транзакций.** Решается использованием ЭЦП транзакций и временных меток.
  4. **Проблема атаки 51%.** Если кто-либо сможет контролировать большинство узлов сети блокчейн, то он сможет контролировать процесс валидации транзакций и цепочек блоков. Решается увеличением числа “честных узлов”, за счет установления правил, делающих “нечестное поведение” узлов не выгодным.

**Inputs** — это транзакции, которые переводят средства на узел. Представим, что на ваш адрес X когда-то было отправлено три транзакции:

TXN\_ID — 123456, VALUE — 40 BTC

TXN\_ID — 6453795, VALUE — 10 BTC

TXN\_ID — 888888, VALUE — 100 BTC

Если нужно потратить, например, 45 BTC, то вы можете сослаться на транзакцию 888888, или сразу на две транзакции: 123456 и 6453795.

**Outputs** — дословно "выходы". Пока что можете считать, что это адреса (хотя это не так), на которые в результате исполнения транзакции будут "отправлены" средства. Выходов также может быть несколько, и каждому из них указывается своя сумма.

Например, новая транзакция C, которая ссылается на два выхода — A и B. В результате на входе у транзакции получается 0.008 BTC, которые потом разделяются на два выхода — на первый адрес отправляется 0.003 BTC, а на второй 0.004 BTC.

## **61. Блоки данных Blockchain. Структура блока. Узлы spv (упрощенной проверки платежей). Упрощенная проверка платежей. метки времени (timestamp) и Цепочка блоков.**

Блоки данных в блокчейне Биткойна состоят исключительно из данных об операциях (транзакциях) с биткойнами. Это огромная записанная история всех когда-либо произведённых транзакций с бикойнами, начиная с самой первой сделки, которая состоялась 3 января 2009 г.

Каждый блок имеет размер приблизительно 1 мегабайт (Мб) данных. При минимальном объеме информации в транзакции около 200 байт, максимально в блоке может быть около 6000 транзакций.

Каждый блок подписан с помощью хэша формируемого с помощью функции SHA-256, который рассчитывается с заданной сложностью, что требует выполнения большой вычислительной работы методом “грубой силы”. Именно поэтому за выполнение этой работы по созданию новых блоков узлы получают оплату в виде комиссионных (после 9 мая 2020 это 6,25 BTC).

Блоки связываются друг с другом, за счет того, что подпись (хэш) каждого последующего блока вычисляется с учетов значения хэша предыдущего блока. Это справедливо для всех блоков, кроме нулевого (0), у которого хэш предыдущего блока отсутствует. Связь блоков в цепочку с помощью ЭЦП обеспечивает неизменность всего реестра транзакций блокчейна, который контролируется всеми узлами сети. Это означает, что изменения одного блока потребует создания новой подписи для каждого следующего за ним блока, и так до самого конца цепочки. На практике такое практически невыполнимо.

## **62. Протокол Bitcoin. Поиск узлов в сети Blockchain. Согласование версий сообщений. Установка соединений. Создание и подпись транзакции. Отправка транзакций.**

**Протокол бк включает в себя следующие компоненты и процедуры:**

* Формирование ключей и адресов
* Поиск узлов в сети Blockchain
* Согласование версий сообщений
* Установка соединений
* Выполнение транзакций
* Подписывание транзакций
* Отправка транзакций

Далее рассмотрим некоторые из них.

**поиск узлов в сети бч**

Адрес формируется на основе открытого ключа узла. Способ формирования в описании криптографии Bitcoin.

Для поиска узлов в сети blockchain используется несколько методов:

* Хранение списка IP узлов, составленного на основе предыдущих подключений к сети. Хорошо работает при регулярном подключении узла к сети. Но плох если узел впервые или редко подключается к сети.
* Использование DNS seeds. То есть использование некоторые доверенные сервера на которых хранятся адреса серверов трекеров, с которых можно получить адреса для доступ к узлам сети blockchain. Это такие адреса: bitseed.xf2.org, dnsseed.bluematt.me, seed.bitcoin.sipa.be, dnsseed.bitcoin.dashjr.org, seed.bitcoinstats.co

Имена сидов жестко зашиты в модуле chainparams.cpp. Их адреса можно легко получить с помощью утилит nslookup и dig.

В учебных целях можно использовать такой метод:

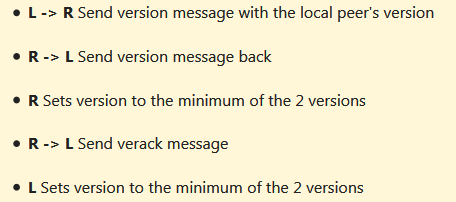
Заходим на Shodan, регистрируемся, авторизуемся

в строке поиска пишем port:8333. Это стандартный порт для bitcoind, будет выведен список узлов, прослушивающих данный порт.

**согласование версий**  
Установка соединения между нодами начинается с обмена двумя сообщениями. Первым отправляется version message, а в качестве ответа на него используется verack message.

Вот иллюстрация процесса version handshake из Bitcoin wiki:

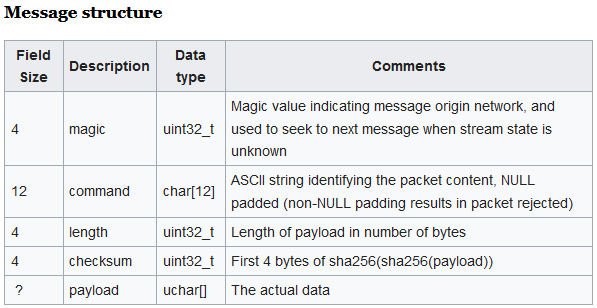
(L-локальный узел, R-удаленный)



Это делается в первую очередь для того, чтобы ноды узнали, какой версией протокола пользуется их "собеседник" и могли общаться на одном языке.

**установка соединений**

Каждое сообщение в сети должно представляться в виде структуры: - magic + command + lenght + checksum + payload, за это отвечает функция makeMessage.



Этой функцией мы еще воспользуемся, когда будем отправлять транзакцию.

Если все сделано верно, то, во-первых, будет верно определен протокол, user-agent, block start height и так далее. Во-вторых, как и обещалось, прилетит ответ в виде сообщений version и verack. Теперь, когда соединение установлено, можно начинать работу.

**создание и подпись транзакции**

Транзакция создается в соответствии со спецификацией, определяющей ее структуру.

После создания транзакция подписывается.

**Замечание.** В протоколе Bitcoin есть уязвимость, связанная с подписью транзакции при ее создании. Эта уязвимость называется Transaction malleability.

**отправка транзакций**

Если все сделано верно, то в качестве ответа на ваше сообщение придет inv message (в противном случае был бы reject message).

Интересный факт — каждый узел, при получении свежей транзакции проверяет ее на валидность, поэтому если вы где-то ошиблись, то вас об этом мгновенно оповестят.

Уже через несколько секунд после отправления транзакции в сеть, ее можно будет отследить, правда сначала она будет числиться неподтвержденной.

Потом, спустя какое-то время (вплоть до нескольких часов), транзакция будет включена в блок.

## **63. Майнинг (добыча) монет bitcoin. Денежная масса bitcoin. Специальная (Первая) транзакция в блоке. Доказательство выполненной работы. Разветвления (forks) цепочек блоков. Консенсус узлов при подтверждении цепочек блоков. Подтверждение блоков и транзакций. Сложность хеша.**

**Майнинг, также добыча** — деятельность по поддержанию распределенной платформы и созданию новых блоков с возможностью получить вознаграждение в форме эмитированной валюты и комиссионных сборов в различных криптовалютах, в частности в Биткойн. Производимые вычисления требуются для обеспечения защиты от повторного расходования одних и тех же единиц валюты, а связь майнинга с эмиссией стимулирует людей расходовать свои вычислительные мощности и поддерживать работу сетей — Wikipedia.

Майнинг — это критически важный для Bitcoin процесс, состоящий в создании новых блоков и преследующий сразу две цели.

* Первая — производство денежной массы. Каждый раз, когда майнер создает новый блок, ему за это полагается награда в N-ое число монет, которые он потом где-нибудь тратит, тем самым запуская в сеть новые средства.
* Вторая, и куда более важная цель, — обеспечение работы всей сети.

**денежная массса бк**  
Одна из фундаментальных фишек, которой часто бравируют сторонники криптовалют — заложенная изначально дефляция. Это связано с тем, что еще на этапе проектировки системы, было указано суммарное ограничение в 21 миллион монет (примерно), и даже если очень сильно захотеть, поднять этот порог не получится. В отличие от рубля или доллара, которые по желанию казначейства могут быть напечатаны в любом количестве, что иногда приводит к печальным последствиям.

В 2140 году награда за блок станет 0 BTC.

Для усложнения процесса добычи монет Сатоши придумал алгоритм, со следующими свойствами:

Создание нового блока — вычислительно сложная задача. Нельзя вот так просто включить мощный ПК и за минуту намайнить сто блоков.

На вычисление нового блока у всей сети уходит 10 минут (в среднем). Если посмотреть на Litecoin, то там блоки выходят раз в 2-3 минуты, суть заключается именно в том, что среднее время заранее установлено.

Более того, это время не зависит от числа участников сети. Даже если однажды майнеров станет в сто раз больше, то алгоритм должен так изменить свои параметры, чтобы блок стало находить сложнее, и block time опустился обратно в окрестность десяти минут.

Помним, что сеть распределенная и одноранговая, а значит, она должна сама понимать, в какой момент и как нужно подкрутить эти параметры. Никаких управляющих узлов, все полностью автономно.

Если решение задачи по созданию нового блока — это сложная задача, требующая времени и ресурсов, то проверка блока на "корректность" должна быть простой и практически мгновенной.

**Специальная (Первая) транзакция в блоке**

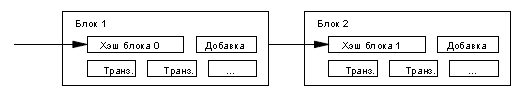
В каждом блоке первая транзакция в списке является особой транзакцией. Новые блоки создаются в системе Bicoin в среднем каждые 10 минут. У нее всегда один вход, у которого вместо свойства scriptSig есть свойство coinbase.

Выход у транзакции также всегда один. Он перенаправляет 50 монеток (после 09.05.2020 6,25 монет) тому, кто сгенерировал блок, в котором расположена эта транзакция. Это своего рода награда за потраченное время и ресурсы на генерацию блока. Создавая новый блок в цепочке, клиент вносит вклад в работу Bitcoin.

Каждые четыре года эта награда уменьшается вдвое, таким образом суммарное количество монеток в обороте стабилизируется. После этого даже если злоумышленник создаст блок с помощью модифицированной версии программы и припишет себе 50 монеток, этот блок не будет добавлен в цепочку, так как его отвергнут другие честные клиенты, которых должно быть большинство.

Стабильность работы системы основана на количестве пользователей, у которых запущен официальный клиент. Пока их большинство, Bitcoin ничего не угрожает.  
Каждый хэш блока данных должен соответствовать установленным условиям сложности вычислений. В Bicoin управление сложностью вычислений основано на системе Hashcash Адама Бека. Суть которой заключается в поиске такого значения данных, чей хэш (например, SHA-256) начинался бы с некоторого числа нулевых битов.

Поиск нужного значения хэша SHA-256 для блока данных выполняется путем перебора значений, размещаемых в дополнительном поле - nonce (добавка):



Для этого требуется выполнить объем работы, экспоненциально зависящий от числа нулей, но для проверки найденного значения достаточно вычислить лишь один хэш.

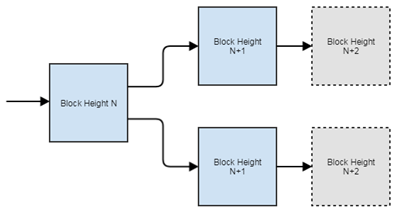
Успешное нахождение хэша и является доказательством проделанной работы для сетей Bitcoin.

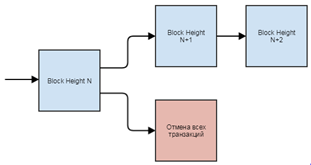
Название достаточно точно определяет суть процесса, т.к. происходит простой перебор вариантов, и если кто-то нашел подходящий хеш, то это действительно удача. Это как найти реальный самородок золота в тоннах пустой породы. Вознаграждение за блок сейчас составляет 12.5 BTC (было до 09.05.2020) и если умножить на актуальный курс биткойна $3900, то получается больше килограмма чистого золота.

**Разветвления (forks)**

Возможен случай, если, например, несколько узлов нашли хеши удовлетворяющие условиям сложности, но разные по значению (иными словами, пришли к разным консенсусам) и записали блоки в блокчейн.

В этом случае происходит, так называемый, форк (‘развилка'), и блокчейн имеет две версии цепочки:



Далее часть сети начинает работать над блоком N+2 от одной цепочки, а часть над блоком от от другой:   


Какой-то из этих блоков будет найден раньше тот будет включен в цепочку и тогда по правилам узел должен будет переключиться на более длинную цепочку и отменить все транзакции из альтернативного блока.

Доказательство работы через хэширование блоков решает вопрос об определении версии блока, поддерживаемой большинством.

Если голосом считается один IP-адрес, то такую схему можно скомпрометировать, только если контролировать большой диапазон адресов.

Схема подтверждения Bitcoin основана на принципе «один процессор — один голос». Самая длинная из хэш-цепочек выражает мнение большинства, которое вложило в нее наибольшее количество ресурсов.

Если более половины вычислительной мощи принадлежит “честным” узлам, то цепочка “честных” транзакций будет расти быстрее и опередит любую конкурирующую цепь. Чтобы внести изменения в любой из прошлых блоков, атакующему придется выполнить заново работу над этим блоком и всеми последующими, а затем догнать и перегнать честных участников по новым блокам.

Можно показать, что вероятность такого успеха у злоумышленника, обладающего меньшими ресурсами, экспоненциально убывает в зависимости от числа блоков.

**подтверждение блоков и транзакций**  
Пока транзакция не включена в блок, система считает, что количество биткойнов на некоем адресе остаётся неизменным. В это время есть техническая возможность оформить несколько разных транзакций по передаче с одного адреса одних и тех же биткойнов разным получателям. Но как только одна из подобных транзакций будет включена в блок, остальные транзакции с этими же биткойнами система будет уже игнорировать.

Попадание транзакции в блок является подтверждением её достоверности вне зависимости от наличия других транзакций с теми же биткойнами. Каждый новый блок считается дополнительным «подтверждением» транзакций из предыдущих блоков. Если в цепочке 3 блока, то транзакции из последнего блока будут подтверждены 1 раз, а помещённые в первый блок будут иметь 3 подтверждения. Достаточно дождаться нескольких подтверждений, чтобы вероятность отмены транзакции стала очень низкой.

Для уменьшения влияния подобных ситуаций на сеть существуют ограничения на распоряжение только что полученными биткойнами. Согласно сервису blockchain.info, до мая 2015 года максимальная длина отвергнутых цепочек была 5 блоков. Клиент «Bitcoin-qt» для отправки не требует наличия подтверждений, но у большинства получателей по умолчанию выставлено требование 6 подтверждений, то есть реально воспользоваться полученным обычно можно через час.

Биткойны, полученные за создание блока, протокол разрешает использовать после 100 подтверждений, но стандартная программа-клиент показывает комиссию через 120 подтверждений, то есть обычно воспользоваться комиссией можно примерно через 20 часов после её начисления.

**сложность хэша**

Для компенсации возрастающей вычислительной мощи процессоров и колебания числа работающих узлов в сети, сложность хэширования должна изменяться, чтобы обеспечивать равномерную скорость генерации блоков. Если они появляются слишком часто — сложность возрастает, и наоборот.

Сложность определятся следующим условием:

* найденный хеш должен быть меньше заранее заданного числа (задается число лидирующих 0 хэша).
* Если данное условие не выполняется, то к полю nonce заголовка блока прибавляется 1, и работа по вычислению хеша повторяется.

## **64. Правила работы узлов консенсуса. Стимулы соблюдения узлами сети Blockchain правил. Дерево Меркла и экономия дискового пространства. Упрощенная проверка платежей. рассылка сигналов тревоги при получении «ложного» блока. Соединение и разделение сумм. Конфиденциальность. Жизненный цикл транзакции Blockchain.**

**Правила работы узлов консенсуса**  
Участники всегда считают истинной самую длинную версию цепочки и работают над ее удлинением.

Если два узла одновременно опубликуют разные версии очередного блока, то кто-то из остальных пиров получит раньше одну версию, а кто-то — другую. В таком случае каждый начнет работать над своей версией цепочки, сохранив другую на случай, если она окажется продолжена раньше.

Двойственность исчезнет, как только будет получен новый блок, который продолжит любую из ветвей, и те узлы, что работали над конкурирующей версией, переключатся на нее.

Новые транзакции не обязательно должны достигать всех узлов. Если о них будет знать достаточно много узлов, вскоре они попадут в один из блоков.

Правила рассылки блоков тоже не являются строгими в отношении потерянных сообщений. Как только узел, пропустивший один из блоков, получит уже следующий за ним, он запросит недостающую информацию, чтобы заполнить очевидный пропуск.

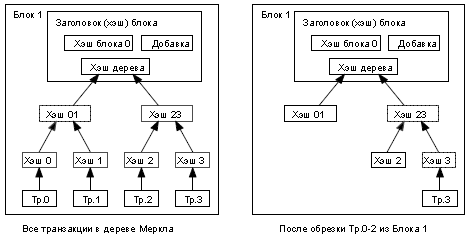
**Стимулы обеспечивают соблюдение правил участниками сети Blockchain**

* **Стимул 1.** По умолчанию, первая транзакция в блоке является специальной, создающей новую монету, которая принадлежит создателю блока. Такая схема поощряет честных участников сети, стимулируя их поддерживать работу сети, а также решает вопрос о начальном распределении денежной массы в отсутствие центрального эмитента.
* **Стимул 2.** Другим способом стимулирования может быть комиссия за транзакции. Если входная сумма платежа больше выходной, то разница является комиссией за перевод и прибавляется к базовому значению награды за найденный блок в первой транзакции. Как только суммарный объем денежной массы достигнет заранее установленного максимума, единственным источником поощрения работы над блоками останутся комиссии, при этом избавленные от инфляции.
* **Стимул 3.** Если жадный злоумышленник способен выделить больше вычислительных мощностей, чем все честные участники, то он сможет обманывать продавцов, аннулируя свои транзакции и возвращая средства на свой счет (вариант «саботажа системы»). Но он может направить свои ресурсы на генерацию новых блоков и монет. Более выгодным для него является вариант «игры по правилам», который обеспечивает получение более половины всех новых денег, чем вариант «саботажа системы» и поддержания своего капитала на постоянном уровне.

**Экономия дискового пространства**

Как только последняя транзакция в монете-цепочке окажется внутри достаточно старого блока, все предшествующие ей транзакции в цепочке могут быть удалены в целях очистки дискового пространства. Чтобы хэш блока остался неизменным, все транзакции в блоке хранятся в виде хэш-дерева Меркла и лишь его корень включается в хэш блока.

Размер старых блоков может быть уменьшен за счет удаления ненужных ветвей этого дерева, хранить промежуточные хэши необязательно.



Заголовок пустого блока будет составлять около 80 байт. Из расчета скорости генерации блока раз в десять минут получаем 80\*6\*24\*365=4.2 Мб в год. Для cреднестатистического на 2008 год компьютера с 2 Гб оперативной памяти с учетом закона Мура, предсказывающего рост на 1.2 Гб в год, хранение данных не будет проблемой, даже если все заголовки блоков будут находиться в памяти.

**рассылка сигналов тревоги при получении «ложного» блока**.

Такой сигнал будет заставлять программу-клиент загружать блок полностью, чтобы самостоятельно подтверждать некорректность данных.

Компании, часто принимающие платежи, возможно, будут подключаться к сети в обычном режиме для большей независимости, безопасности и быстроты проверки.

**Соединение и разделение сумм**

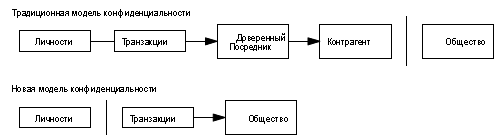
Несмотря на то, что можно оперировать отдельными монетами, создавать специальную транзакцию для каждого цента было бы слишком неудобно. Для поддержки разделямых и объединяемых сумм транзакции содержат несколько входов и выходов.

Обычная транзакция будет выглядеть так: либо один вход от предыдущего крупного платежа, либо несколько входов, аккумулирующих небольшие суммы, и не более двух выходов: один является собственно платежом, а другой, если необходимо, возвращает «сдачу» обратно отправителю.

**конфиденциальность**

Традиционная банковская модель поддерживает необходимый уровень конфиденциальности, предоставляя доступ к информации лишь сторонам-участницам и доверенному третьему лицу. Необходимость открытой публикации транзакций исключает такой подход, однако приватность по-прежнему можно сохранить, если публичные ключи будут анонимными.

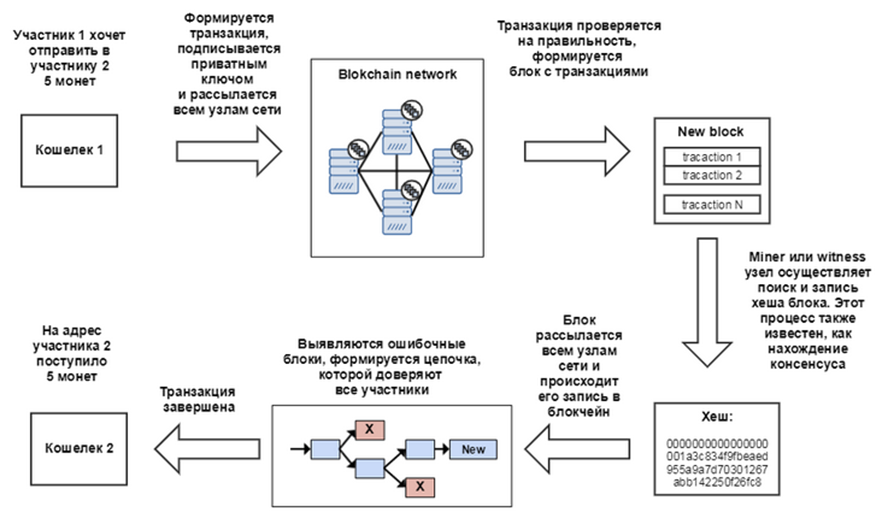
Открытой будет информация о том, что кто-то отправил кому-то некоторую сумму, но без привязки к конкретным личностям. Аналогичным образом ведется работа на фондовых биржах, которые публикуют время и объем частных сделок, не указывая, между кем именно они были совершены.



Дополнительной защитой будет являться генерация новой пары «открытый/закрытый ключ» для каждой транзакции: это предотвратит связывание различных платежей с их общим отправителем или адресатом.

Некоторого публичного связывания все же не избежать: транзакции с несколькими входами доказывают, что эти суммы принадлежат одному лицу. Риск состоит в том, что раскрытие личности владельца ключа может привести к раскрытию и всех принадлежащих ему транзакций.

**жизненный цикл**



# **Тема 9. Процессы и потоки в РИС**

## **65. Реализация взаимодействия между компонентами в распределенных системах. Мультипрограммирование или многозадачный режим работы. Мультипрограммирование в системах пакетной обработки и системах разделения времени.**

В распределенных системах взаимодействие между компонентами реализуется средствами операционных систем узлов.

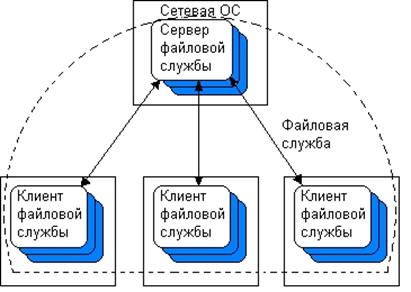
Основным элементом с помощью которого реализуются функции, выполняемые узлами, являются процессы.

Концепция процесса зародилась в операционных системах, где под этим понятием обычно обозначают выполняемую программу.

С точки зрения ОС наиболее важными являются вопросы планирования и управления процессами.

В РС наиболее важными являются вопросы:

* скорость реакции узла на поступающие запросы;
* минимизации времени ответа;
* обеспечение прозрачности доступа к ресурсам РС;
* масштабируемость сервиса при возрастании потока запросов и т.п

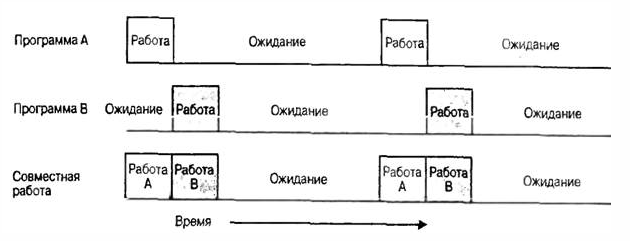


В современных ОС для повышения производительности и скорости выполнения пользовательских задач применяют режимы мультипрограммирования и разделения времени.

Режим мультипрограммирования – способ организации вычислительного процесса, при котором в памяти компьютера находилось одновременно несколько программ, попеременно выполняющихся на одном процессоре. Можно так же определить мультипрограммирование как режим работы вычислительной системы, когда один процесс ожидает освобождения необходимого для него ресурса, а другой процесс в это же время занимает ресурсы процессора. Суть этого режима можно проиллюстрировать на следующем примере.

В системе, работающей на последовательном процессоре, имеются два процесса готовых к выполнению: процесс А и процесс В.

В тот момент, когда процессу А необходимо осуществить операцию ввода/вывода, процесс В занимает ресурс центрального процессора CPU.



Режим разделения времени, как вариант многозадачного режима работы вычислительной системы сводится к следующему:

- каждый процесс имеется свое пространство ресурсов;

- CPU обменивается короткими командами с запущенными процессами;

- операционная система осуществляет передачу команд между CPU и процессами и управляет очередностью этого обмена.

**Многопользовательский режим работы. Режимы реального времени.**

Многопользовательский режим работы вычислительных систем характеризуется, так же, как и мультипрограммный наличием своего пространства ресурсов у каждого процесса, наличием способов обмена сообщениями и управления таковым на уровне операционной системы. Дополнительные возможности предоставляются за счет распределения всех ресурсов вычислительной системы не только между процессами одного пользователя, но и между сеансами управления разных одновременно присутствующих в системе пользователей.

В современных операционных системах, для улучшения качества и скорости выполнения некоторых типов процессов управление производится в режиме реального времени. Сущность данного способа управления вычислительной системой в обеспечении обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать.

## **66. Понятие процесса выполнения. Составляющие контента процесса. Жизненный цикл процесса в ОС. Системные вызовы управляющие процессами. Смена режимов работы процессора при выполнении системного вызова. Виды системных вызовов, связанных с процессами. Идентификация процессов в системе. Дерево процессов.**

**Процессы выполнения**

Для выполнения программ операционная система создает несколько виртуальных процессоров, по одному для каждой программы.

Чтобы отслеживать эти виртуальные процессоры, операционная система поддерживает таблицу процессов, содержащую записи для сохранения значений регистров процессора, карт памяти, открытых файлов, учетных записях пользователей, привилегиях и т. п.

Процесс часто определяют как выполняемую программу, то есть программу, которая в настоящее время выполняется на одном из виртуальных процессоров операционной системы. Это массив (или связанный список) структур хранящий информацию о процессах исполняемых системой.

Каждому процессу в таблице процессов соответствует один экземпляр структуры, хранящий информацию о данном процессе.

Процесс может находиться в одном из двух состояний:

* выполнение;
* остановлен.

жизненный цикл процесса в ос включает в себя следующие стадии:

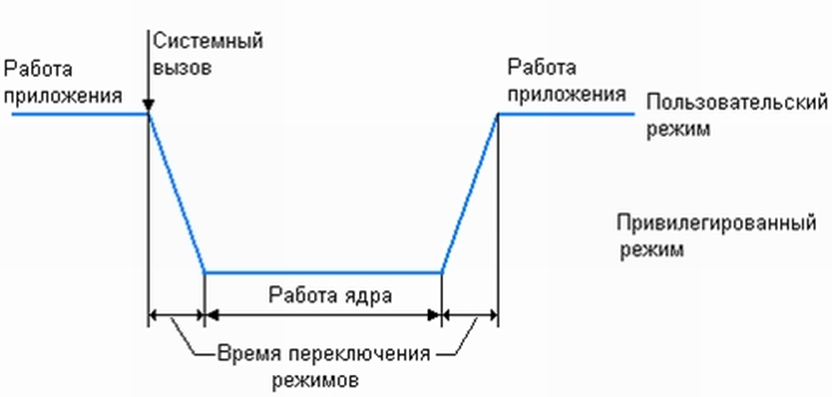
* Создание процесса;
* выполнение процесса;
* уничтожение процесса.

**Системные вызовы управляющие процессами**

Процесс создается родительским процессом с помощью обращения к функции ядра «создать процесс». Обращение к функциям ОС называется **системным вызовом.**

Главными системными вызовами, управляющими процессами, являются вызовы, связанные с созданием и уничтожением процессов. Системный вызов выполняется в привилегированном режиме работы процессора.

**Смена режимов работы процессора при выполнении системного вызова**



**Виды системных вызовов связанных с процессами**

* создание процесса;
* освобождение или выделение дополнительной памяти процессу;
* ожидание завершения какого-либо процесса;
* перекрытие адресных пространств процесса;
* передача сигнала процессу;
* завершение процесса; и др.

**Идентификация процессов в системе**

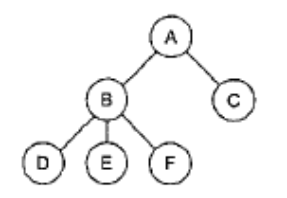
Для идентификации процессов в системе используются идентификаторы процессов PIDs

PID – это число, присваиваемое процессу при запуске.

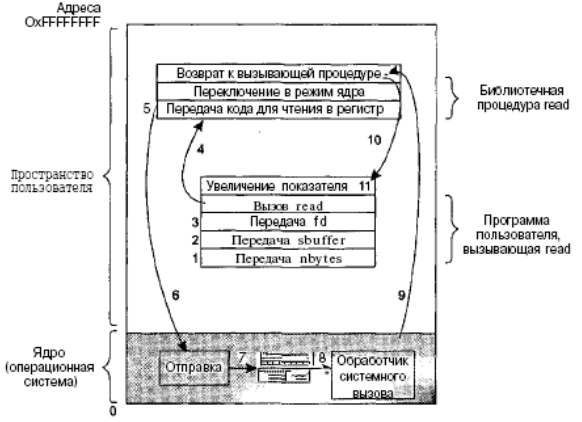
Каждому процессу присваивается идентификатор пользователя, запустившего данный процесс.

**дерево процессов**

Если процесс может создавать несколько других процессов (называющихся дочерними процессами), а эти процессы, в свою очередь, тоже могут создать дочерние процессы, таким образом формируется дерево процессов,



## **67. Выполнение системных вызовов в ОС Unix.**



1-3 размещение параметров вызова в стек;

4 - выполнение вызова библиотечной процедуры read;

5 – размещение номера системного вызова в регистр;

6 – переключение в режим ядра;

7 – проверка номера системного вызова и передача управления обработчику;

8 – выполнение обработчика;

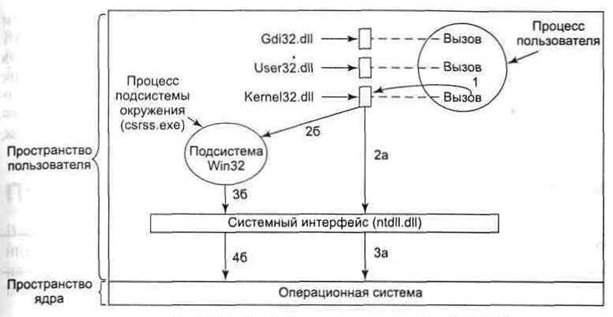
9 – возврат в режим пользователя;

10 – возврат из процедуры read;

11 – очистка стека (восстановление указателя вершины стека).

count = read(fd, buffer, nbytes);

## **68. Выполнение вызовов Win32 API в ОС Windows NT/2kX.**



Вариант 1 (короткий путь).

2а и За динамические библиотеки обращаются к другой динамической библиотеке (ntdll.dll) которая, в свою очередь, обращается к ядру операционной системы. При этом dll может выполнить всю работу самостоятельно, совсем не обращаясь к системным вызовам;

Вариант 2 (длинный путь).

2б, 3б и 4б. Для других вызовов Win32 API выбирает маршрут: процесс пользователя обращается к подсистеме Win32 (csrss.exe), которая выполняет некоторую работу, и затем обращается к системному вызову (ntdll.dll);

В первой версии Windows NT практически все вызовы Win32 API выбирали маршрут 2б, 3б, 4б, так как большая часть операционной системы (например, графика) была помещена в пространство пользователя.

Начиная с версии NT 4.0, для увеличения производительности большая часть кода была перенесена в ядро (в драйвер Win32/GDI);

В Windows 2000 и всех последующих версиях WinNT только небольшое количество вызовов Win32 API, например вызовы для создания процесса или потока, идут по длинному пути. Остальные вызовы выполняются напрямую, минуя подсистему окружения Win32.

## **69. Связанные процессы и межпроцессное взаимодействие. Взаимоблокировка процессов. Методы межпроцессного взаимодействия. Сигналы, передаваемые процессам. Семафоры и мьютексы. Недостатки процессов.**

**Связанные процессы и межпроцессное взаимодействие**

**Связанные процессы** — это те, которые объединены для выполнения некоторой задачи, и им нужно часто передавать данные от одного к другому и синхронизировать свою деятельность. Такая связь называется **межпроцессным взаимодействием** — это набор методов для обмена данными между потоками процессов. Процессы могут быть запущены как на одном и том же компьютере, так и на разных, соединенных сетью.

Когда взаимодействуют два или более процессов, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи. Такая ситуация называется тупиком, тупиковой ситуацией или взаимоблокировкой.

**Методы межпроцессного взаимодействия**

Существуют следующие методы межпроцессного взаимодействия:

* **Каналы (pipe)** средство связи стандартного вывода одного процесса со стандартным вводом другого. Каналы старейший из инструментов IPC, существующий приблизительно со времени появления самых ранних версий оперативной системы UNIX.
* **Сигналы.** Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении, например SIGFPE (ошибка вычислений с плавающей запятой) и SIGSEGV (ошибка адресации), но большинство сигналов является асинхронными.
* **Очереди сообщений**. Очереди сообщений представляют собой связный список в адресном пространстве ядра. Очереди сообщений как средство межпроцессной связи позволяют процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемыми сообщениями. Процессы, использующие этот тип межпроцессной связи, могут выполнять две операции: послать или принять сообщение.
* **Семафоры и мьютексы.** Семафор – это целая переменная, значение которой можно опрашивать и менять только при помощи неделимых (атомарных) операций. Двоичный семафор может принимать только значения 0 или 1. Вычислительный семафор может принимать целые неотрицательные значения. Мьютекс - отличается от семафора тем, что только владеющий им поток может его освободить.
* **Разделяемая память.** Разделяемая память может быть наилучшим образом описана как отображение участка (сегмента) памяти, которая будет разделена между более чем одним процессом. Это гораздо более быстрая форма IPC, потому что здесь нет никакого посредничества (т.е. каналов, очередей сообщений и т.п.).
* **Сокеты.** Сокеты обеспечивают двухстороннюю связь типа точка-точка между двумя процессами. Они являются основными компонентами межсистемной и межпроцессной связи. Каждый сокет представляет собой конечную точку связи, с которой может быть совмещено некоторое имя. Он имеет определенный тип, и один процесс или несколько, связанных с ним процессов.

**Сигналы, передаваемые процессам**

Сигналы являются программными аналогами аппаратных прерываний и могут быть сгенерированы по различным причинам, а не только из-за истечения какого-либо интервала времени.

Многие аппаратные прерывания (например, вызванные выполнением недопустимой команды или использованием неправильного адреса) также преобразуются в сигналы процессу, в котором произошла ошибка.

Сигнал вызывает:

* временную остановку работы процесса независимо от того, что процесс делает в данный момент;
* сохраняет его регистры в стеке
* запускает специальную процедуру обработки сигнала (например, передающую повторно предположительно потерянное сообщение).

После завершения обработки сигнала работающий процесс запускается заново в том состоянии, в котором он находился до сигнала.

**Семафо́р** — примитив синхронизации работы процессов и потоков, в основе которого лежит счётчик, над которым можно производить две атомарные операции: увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся. Служит для построения более сложных механизмов синхронизации и используется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

**Мью́текс** — аналог одноместного семафора, служащий в программировании для синхронизации одновременно выполняющихся потоков. Мьютекс отличается от семафора тем, что только владеющий им поток может его освободить, т.е. перевести в отмеченное состояние. Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

**Недостатки процессов**

Хотя процессы являются строительными блоками распределенных систем, практика показывает, что дробления на процессы, предоставляемого операционными системами, на базе которых строятся распределенные системы, недостаточно.

Вместо этого оказывается, что наличие более тонкого дробления в форме нескольких потоков выполнения на процесс значительно упрощает построение распределенных приложений и позволяет добиться лучшей производительности.

## **70. Понятие потока выполнения. Преимущества потоков выполнения. Реализация потоков на пользовательском уровне и в ядре. Преимущества потоков выполнения на пользовательском уровне. Преимущества использования потоков против группы конкурирующих процессов.**

**Понятие потока выполнения**

Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически быть заблокированной.

Модель программирования упрощается за счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме.

**Реализация потоков выполнения**

Потоки выполнения обычно существуют в виде пакетов. Подобные пакеты содержат механизмы для создания и уничтожения потоков, а также для работы с переменными синхронизации, такими как мьютексы и условные переменные.

Существует два основных подхода к реализации пакетов для потоков выполнения.

* Первый из них состоит в создании библиотеки для работы с потоками выполнения, выполняющейся исключительно в режиме пользователя.
* Второй подход предполагает, что за потоки выполнения отвечает (и управляет ими) ядро.

**Преимущества потоков выполнения на пользовательском уровне (1)**

Библиотека для работы с потоками выполнения на пользовательском уровне имеет множество преимуществ.

Во-первых, дешевле обходится создание и уничтожение потоков выполнения. Поскольку все управление потоками реализуется в адресном пространстве пользователя, стоимость создания потока выполнения определяется в первую очередь затратами на память, выделяемую для создания стека под поток.

Аналогично и уничтожение потока выполнения в основном состоит в освобождении памяти, задействованной под стек, после того как надобность в потоке отпадает. Обе операции достаточно дешевы.

Второе преимущество потоков выполнения на пользовательском уровне состоит в том, что переключение контекста требует всего нескольких инструкций.

В основном в сохранении и последующем восстановлении сохраненных значений при переключении с потока на поток нуждаются исключительно в операциях со значениями регистров процессора.

Нет необходимости изменять карты памяти, сбрасывать буфер TLB, контролировать загрузку процессора и т. д.

Переключение контекста потоков выполнения производится при необходимости в синхронизации двух потоков, например, при обращении к секции совместно используемых данных.

**Реализация потоков в ядре**

Для потоков, реализуемых на уровне ядра, вся информация о потоках аналогична той, что используется для пользовательских потоков.

При работе в режиме ядра создание и уничтожение потоков требует более существенных затрат.

Хотя потоки на уровне ядра могут решить многие проблемы, но их главный недостаток в весьма существенных затратах на реализацию системных вызовов, поэтому если потоки создаются/удаляются достаточно часто, то это влечет за собой существенные издержки.

Другой проблемой являются сигналы. Сигналы предназначены для процессов, а не для потоков.

**Использование потоков против применения группы конкурирующих процессов**

Применение потоков является способом одновременного и параллельного исполнения в рамках одного приложения.

На практике, часто можно встреть реализации, когда приложения строятся как коллекция параллельно работающих процессов, объединяемых с помощью средств межпроцессного взаимодействия предлагаемых операционными системами.

Примером применения такого подхода является реализация веб-сервера Apache, который по-умолчанию стартует как пятерка процессов, предназначенных для обслуживания поступающих запросов.

Использование процессов вместо потоков имеет одно важное преимущество – обеспечивается разделение пространств данных процессов.

Разделение процессов и их пространств данных обеспечивается аппаратными средствами процессоров. Это преимущество нельзя недооценивать, так как при использовании потоков вся забота о параллельном доступе к разделяемым данным ложиться на разработчика многопоточного приложения.

## **71. Потоки в распределенных системах. Зачем потоки используют в РИС. Многопоточные клиенты. Многопоточные серверы.**

**Почему потоки применяют в РИС**

Важнейшим свойством потоков выполнения является возможность выполнения системных блокирующих вызовов, без блокировки всего процесса в рамках которого работает поток.

Это свойство является привлекательным для распределенных систем, что делает их более приспособленными к быстрым коммуникациям и обеспечивает одновременное поддержание нескольких логических соединений между компонентами РС.

**Многопоточные клиенты**

Для обеспечения высокой степени прозрачности клиенты должны обладать поддержкой работы в многопоточном режиме.

Например, Web-браузер, для сокрытия достаточно больших величин задержек должен поддерживать возможность поддержания нескольких одновременных соединений с веб-вервером для одновременной закачки содержимого различных частей веб страниц.

Кроме того, современные браузеры могут одновременного выполнять доставку нескольких веб страниц с различных серверов, что также требует многопоточности.

**Многопоточные серверы**

Рассмотрим файловый сервер. Обычно файловый сервер ожидает обращений от клиентов с запросами на выполнение файловых операций.

Часто такой сервер реализуется в виде нескольких параллельно исполняемых потоков, один из которых выполняет роль диспетчера запросов (поток диспетчер), а остальные реализуют рабочие потоки.

Данная реализация обеспечивает необходимую прозрачность и производительность этого файлового сервера.

# **Тема 10. Виртуализация**

## **72. Понятие виртуализации. Виртуализация ЭВМ. Виртуализация ресурсов физического сервера. Цели виртуализации. Понятие виртуальной машины, ее достоинства и недостатки. Виртуализация и распределенные системы**

В компьютерных технологиях под термином **"виртуализация"** обычно понимается абстракция вычислительных ресурсов и предоставление пользователю системы, которая "инкапсулирует" (скрывает в себе) собственную реализацию. Проще говоря, пользователь работает с удобным для себя представлением объекта, и для него не имеет значения, как объект устроен в действительности.

**Преимущества виртуализации:**

* Эффективное использование вычислительных ресурсов
* Сокращение расходов на инфраструктуру
* Снижение затрат на программное обеспечение.
* Повышение гибкости и скорости реагирования системы.
* Несовместимые приложения могут работать на одном компьютере
* Повышение доступности приложений и обеспечение непрерывности работы предприятия
* Возможности легкой архивации
* Повышение управляемости инфраструктуры

**Виртуализация серверов** — размещение нескольких логических серверов в рамках одного физического.

**Цели виртуализации:**

* Предоставить каждому пользователю изолированную среду исполнения приложений.
* Повысить гибкость использования ресурсов ЭВМ исполняемыми на ней приложениями.
* Повысить защищенность приложений друг от друга, исполняемых на одной и той же ЭВМ.
* Повысить эффективность использования аппаратных средств ЭВМ.

**Достоинства:**

* Эффективность использования ресурсов
* Масштабируемость
* Простые резервное копирование и миграция
* Гибкость

**Недостатки:**

* Проблема в распределении ресурсов при высокой загрузке
* Vendor lockin
* Сложная настройка

## **73. История появления и развития технологии виртуализации.**

1965. Выражение “Hypervisor” впервые появилось применительно к ПО обработки RPQ на ЭВМ IBM 360/65.

Примерно 1966. В кембриджском научном центре разработан эмулятор CP-40 для S/360-40, явившийся первой попыткой реализации полной программной виртуализации физической ЭВМ.

1967. На основе этой разработки была создана CP[-67]/CMS – virtual machine/virtual memory time-sharing operating system для S/360-67

В начале 70-х гипервизор СР-67, был переработан в виде OS VM/370 для нового семейства машин System/370, выпущенного на рынок в 1972 г.

Линейка машин System/370 в 1990-х годах была заменена компанией IBM линейкой System/390. Виртуализация в ОС MVS 390 была сохранена.

В 2000 году IBM выпустила машины z-серии, поддерживающие 64-разрядное виртуальное адресное пространство при сохранении обратной совместимости с System/360 и поддержке виртуализации.

Все эти системы фирмы IBM поддерживали виртуализацию на десятилетия раньше того момента, когда она приобрела популярность на машинах семейства x86.

1980-90 г.г. В это время основные работы в области виртуализации велись в направлении адаптации этой технологии для персональных ЭВМ и прежде всего для архитектуры Intel x86.

В 1999 г. компания VMware представила технологию виртуализации систем на базе x86 получившую название VMware Virtual Platform. Первым продуктом реализующим новую технологию было ПО VMware WorkStation (гирервизор на основе хозяйской ЭВМ).

2005 г. VMware выпустила первое бесплатное ПО виртуализации десктопов - VMware Player.

2006 г. Выпустила ПО VMware ESX Server – первый гипервизор полной виртуализации серверов для архитектуры х86 (native hypervisor – “родной” гипервизор х86)

Позднее в "битву“включились такие компании как:

Parallels (ранее SWsoft), продукты Parallels Workstation, Parallels Desktop для Mac (2006), Parallels Virtuozzo Containers (технология виртуализации средствами ОС);

Oracle (Sun Microsystems), VirtualBox (2008) была приобретена у компании Innotek в 2007;

Citrix Systems (XenSourse), Xen разработан в кембриджеском университете (там же где и CP-40), первый публичный релиз Xen выпущен в 2003. Гипервизор Xen использует технологию паравиртуализации. В октябре 2007 Citrix купила XenSource и осуществила переименование продуктов Xen;

RedHat. Программное обеспечение KVM было создано, разрабатывается и поддерживается фирмой Qumranet, которая была куплена Red Hat за $107 млн 4 сентября 2008 года. После сделки KVM (наряду с системой управления виртуализацией oVirt) вошла в состав платформы виртуализации RHEV;

Корпорация Microsoft вышла на рынок средств виртуализации в 2003 г. с приобретением компании Connectiх, выпустив свой первый продукт Virtual PC для настольных ПК. Microsoft Hyper-V (кодовое имя Viridian), — система аппаратной виртуализации для x64-систем на основе гипервизора. Бета-версия Hyper-V была включена в x64-версии Windows Server 2008, а законченная версия (автоматически, через Windows Update) была выпущена 26 июня 2008.

2011г. Сформулированы основные модели развертывания и признаки облачных вычислений.

В основе облачных вычислений лежат технологии виртуализации.

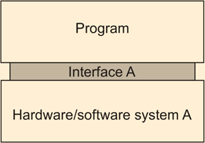
## **74. Принципы виртуализации. Виды интерфейсов в компьютерной системе. Типы виртуализации. Два способа реализации виртуализации. Виртуализация на уровне среды исполнения прикладной программы.**

**принципы виртуализации**

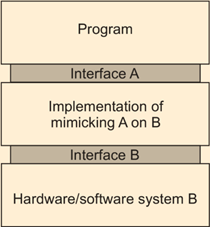
В реальной программной системе имеется ряд интерфейсов, начиная с базового набора команд ЦПУ и кончая коллекцией АРI поставляемых с различным ПО современных систем с промежуточным слоем.

Виртуализация – это по сути замена существующих интерфейсов на интерфейсы, имитирующие поведение других систем.

В компьютерной системе имеется ряд интерфейсов, начиная с базового набора команд ЦПУ и кончая коллекцией АРI поставляемых с различным ПО современных систем с промежуточным слоем.



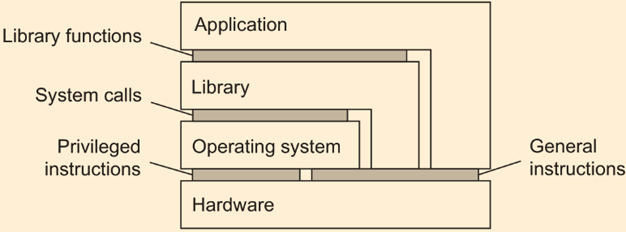
Например, для исполнения под ОС Unix приложений Windows, необходимо использовать эмулятор обеспечивающий преобразование системных вызовов Windows в системные вызовы UNIX.

****

**Виды интерфейсов в компьютерной системе**

В компьютерной системе в общем случае выделяют 4 слоя, связанные тремя типам интерфейсов:

* Интерфейс между аппаратными средствами и операционной системой
* Интерфейс системных вызовов ОС
* Интерфейс вызова библиотечных функций



Тип виртуализации определяются типом интерфейса на котором она выполняется:

* Виртуализация на уровне набора команд (интерфейс ISA). При этом надо учитывать, что в состав набора команд входят привилегированные и непривилегерованные команды;
* Виртуализация на уровне системных вызовов (интерфейс системных вызовов);
* Виртуализация на уровне библиотечных вызовов (на интерфейсе вызова библиотечных функций или API).

**Два способа реализации виртуализации**

Суть виртуализации – имитация поведения интерфейса компьютерной системы. Существуют два способа реализации такой имитации:

* Средствами среды исполнения приложения.
* Средствами экранирования аппаратных средств машины с помощью ПО преобразования в набор команд той же или другой архитектуры, используемого в качестве интерфейса. Это ПО получило название VMM – Virtual Machine Monitor (Монитор виртуальных машин).

**Виртуализация на уровне среды исполнения прикладной программы. Виртуальная машина одного процесса.**

В этом случае среда исполнения обеспечивает либо:

* интерпретацию кода приложения, построенного на основе некоторого абстрактного набора команд. Пример: Java VM, реализующая интерпретацию байт кода в машинные команды аппаратной платформы.
* эмуляцию кода приложения, написанного для одной ОС в код приложения, предназначенного для другой ОС. Пример: ПО Wine для ОС Linux, позволяющее исполнять программы для Windows.

Оба способа исполнения работают только с непривилегированными командами процессора, поэтому они не требуют специальных условий для испонения кода. Такая виртуализация получила название виртуальной машины одного процесса.

## **75. Требования к архитектуре ЭВМ, для поддержки виртуализации. Проблемы виртуализации в архитектуре Intel x86. Монитор виртуальных машин. Два подхода к реализации монитора виртуальных машин.**

**Требования к архитектуре ЭВМ, для поддержки виртуализации**

* Безопасность — у гипервизора должно быть полное управление виртуализированными ресурсами.
* Эквивалентность — поведение программы на виртуальной машине должно быть идентичным поведению этой же программы, запущенной на реальном оборудовании.
* Эффективность — основная часть кода в виртуальной машине должна выполняться без вмешательства гипервизора.

**Проблемы виртуализации в архитектуре Intel x86**

В наборе команд Intel x86 (включая и х64) имеются команды способные изменить состояние процессора исполняемые в пользовательском режиме. Различают команды:

* поведение которых зависит от режима исполнения (behavior sensitive);
* которые влияют на управление (control senitive)

Например команда РОРF, которая устанавливает флаг разрешения прерывания, только когда работает в режиме ядра, но относится к общим командам.

Имеется еще 17 команд являющихся непривилегированными и неперехватываемых операционной системой, но чувствительных к режиму исполнения.

Для решения проблемы виртуализации архитектуры Intel используется дополнительное ПО, получившее название монитор виртуальных машин.

**Монитор виртуальных машин**

Монитор виртуальных машин обеспечивает интерфейс между гостевой ОС и хозяйской ОС. При этом подавляющая часть кода приложения, гостевой ОС, а также хозяйской ОС выполняется напрямую на аппаратных средствах физической машины (все общие команды и часть привилегированных).

И лишь небольшая часть кода, которую составляют некоторые привилегированные команды перехватываются монитором виртуальных машин, модифицируются им, и затем передаются на исполнение аппаратным средствам. Монитор виртуальных машин получил название Гипервизор.

Различают: гипервизоры I и II типов.

**Два подхода к реализации монитора виртуальных машин**

* Эмуляция всех инструкций, что влечет за собой резкое падение производительности. Этот подход в измененном виде используется современными мониторами виртуальных машин.
* Использование паравиртуализации, которая предполагает модификацию гостевых ОС, таким образом, чтобы нейтрализовать, либо полностью исключить влияние “чувствительных” команд, средствами хозяйской ОС.

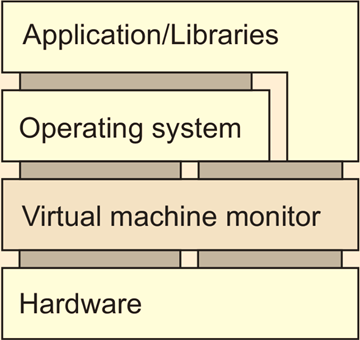
## **76. Монитор виртуальных машин, исполняемый как отдельная ОС (Гипервизор первого типа).**

На аппаратные средства устанавливается специализированная ОС (гипервизор) предназначенная для виртуализации на уровне набора команд аппаратной платформы (Vmware ESX). Такой монитор получил название “родного” (naitive) монитора виртуальных машин.

**Достоинства** данной технологии заключаются в: отсутствии потребности в хостовой ОС – ВМ, устанавливаются фактически на "голое железо", а аппаратные ресурсы используются более эффективно.

**Недостатки:**

* необходимость поддержки в гипервизоре собственных драйверов внешних устройств, из-за чего возникают высокие дополнительные накладные расходы на используемые аппаратные ресурсы,
* отсутствие учета особенностей гостевых ОС, меньшая, чем нужно, гибкость в использовании аппаратных средств



## **77. Монитор виртуальных машин, исполняемый в среде хозяйской ОС (Гипервизор второго типа).**

Монитор виртуальных машин работает в рамках хозяйской ОС.

В процессе своей работы он модифицирует код гостевых ОС с целью обеспечения их виртуализации на уровне набора команд.

При этом код, содержащий “чувствительные” команды переписывается на лету (Vmware WorkStation).

## **78. Паравиртуализация.**

техника виртуализации, при которой гостевые операционные системы подготавливаются для исполнения в виртуализированной среде, для чего их ядро незначительно модифицируется.

Операционная система взаимодействует с программой гипервизора, который предоставляет ей гостевой API, вместо использования напрямую таких ресурсов, как таблица страниц памяти.

Этот подход не только поддерживает высокую производительность, но и позволяет формировать гетерогенную среду, в которой работает несколько гостевых операционных систем.

Накладные расходы составляют 5-10%

## **79. Виртуализация на уровне операционной системы. Контейнерная виртуализация в Linux. Виртуализация против контейнеризации.**

виртуализирует физический сервер на уровне ОС, позволяя запускать изолированные виртуальные серверы называемые Виртуальные Частные Серверы (VPS) или Контейнеры (CT).

Виртуализация на уровне операционной системы имеет минимальные накладные расходы и обеспечивает самую высокую степень консолидации, однако эта технология не позволяет запускать ОС с ядрами, отличными от ядра базовой ОС.

Накладные потери производительности контейнеров составляют 3%

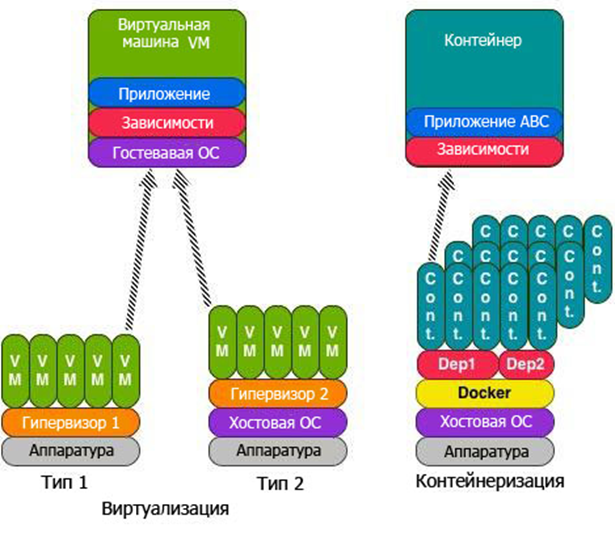
**Контейнерная виртуализация в Linux**

Экземпляры пространств пользователя (часто называемые контейнерами или зонами) с точки зрения пользователя полностью идентичны реальному серверу, но они в своей работе используют один экземпляр ядра операционной системы.

Для linux-систем, эта технология может рассматриваться как улучшенная реализация механизма chroot.

Ядро обеспечивает полную изолированность контейнеров, поэтому программы из раз­ных контейнеров не могут воздействовать друг на друга.

Наиболее распространены сейчас OpenVZ, LXC, FreeBSD jail и Solaris Containers



# **Тема 11. Основы облачных технологий**

## **80. Понятие облачных вычислений. Признаки облачных вычислений. Облачные вычисления – результат синтеза целого ряда технологий и подходов.**

Под **облачными вычислениями** мы понимаем программно-аппаратное обеспечение, доступное пользователю через Интернет или локальную сеть в виде сервиса, позволяющего использовать удобный интерфейс для удаленного доступа к выделенным ресурсам (вычислительным ресурсам, программам и данным).

Признаки облачных вычислений

* Самообслуживание по требованию в условиях мультиарендности.
* Широкий (универсальный и высокоскоростной) сетевой доступ.
* Объединение ресурсов в пулы.
* Мгновенная эластичность ресурсов (мгновенная масштабируемость).
* Измеряемый сервис (учет реального объема потребляемых ресурсов).

Облачные вычисления – результат синтеза целого ряда технологий и подходов

Один облачный сервер обходится дешевле, чем сервер, приобретенный и установленный самой компанией:

* более разумное расходование электроэнергии + размещение дата-центров в регионах с более выгодными тарифами.
* оптимизация использования рабочей силы (один администратор - не 100, а 1000 серверов).
* более высокая безопасность и надежность

Облачные технологии основываются на использовании:

- сервис-ориентированной архитектуры ИС;

- технологий динамического Web (Web 2.0);

- современных систем разработки ПО;

- подходов и методов построения систем масштабируемых

кластерных вычислений;

- методов и средств создания grid-систем;

- технологий виртуализации;

- модель предоставления сервиса IaaS.

## **81. Общая архитектура облачных вычислений.**

Включает следующие компоненты:

* ЦОДы – центры обработки данных;
* платформы снабжения виртуализированными вычислительными, сетевыми ресурсами и ресурсами СХД и СУБД из нескольких ЦОДов для обеспечения запросов от мультиарендных приложений;
* управление снабжением ресурсами облака;
* управление доступом к облаку;
* каталоги предоставляемых облаком сервисов;
* мониторинг производительности и безопасности облака.

**Многоуровневая архитектура облака**

Используется для описания моделей обслуживания пользователей и развертывания

Архитектура облака разработана как 3-х уровневая:

- Уровень инфраструктуры (IaaS);

- Уровень платформы (PaaS);

- Уровень приложений (SaaS).

## **82. Сервисные модели облачных вычислений. Облачные вычисления как эволюция архитектуры корпоративных приложений. Типы облаков. Уровень инфраструктуры (IaaS)**

Служит основой для создания других уровней облака.

Строится на основе виртуализированных вычислительных, сетевых и ресурсов систем хранения.

Виртуализация обеспечивает автоматическое снабжение ресурсами и оптимизирует процесс управления ими.

**Уровень платформы (PaaS)**

Предназначен для общего в том числе и для повторного использования коллекции программных ресурсов.

Это уровень обеспечивает для пользователей среду для разработки своих приложений, для тестирования правильности их функционирования, а также для оценки результатов вычислений и уровня производительности.

Он должен гарантировать пользователям необходимый уровень масштабируемости, независимости и защищенности.

В некотором смысле виртуализированный уровень платформы служит системным промежуточным слоем между инфраструктурным уровнем и уровнем приложений.



**Уровень приложений (SaaS)**

Формируется совокупностью всех программных модулей необходимых для выполнения приложений SaaS.

Сервисные приложения обслуживающие этот уровень включают:

Управление ежедневными офисными работами, такими как, поиск информации, обработка документов, планирование рабочего времени, аутентификация.

Это уровень широко используется для автоматизации деловых процессов, работе с потребителями, выполнения финансовых транзакций, управления отношениями с партнерами и потребителями, управлениями цепочками поставок.

Следует отметить, что не все облачные сервисы ограничиваются только этим одним уровнем. Существует много приложений использующих ресурсы нескольких уровней.



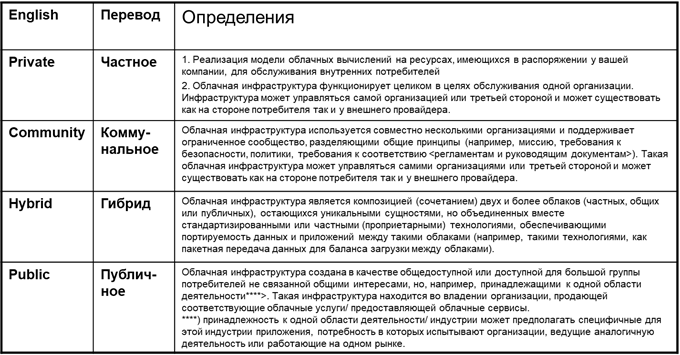
**Облачные вычисления как эволюция архитектуры корпоративных приложений**

Облачные вычисления – это следующий шаг в эволюции архитектуры построения информационных систем.

Стадии эволюции архитектур ИС:

* Монолитная архитектура приложений и целых информационных систем, когда данные и приложения работали на одном компьютере
* Архитектура "клиент-сервер"
* ЦОД - виртуализация ресурсов (серверов, хранилищ данных, сетевого оборудования и клиентских рабочих мест).
* Облачные вычисления – системы позволяющие создавать ИС нужной конфигурации по требованию потребителя.

Типы облаков



Внутреннее частное облако

**«Облачная инфраструктура функционирует целиком в целях обслуживания одной организации. Инфраструктура управляется самой организацией и существует на стороне потребителя» NIST**

Внешнее частное облако

**«Облачная инфраструктура функционирует целиком в целях обслуживания одной организации. Инфраструктура управляется третьей стороной и существует на стороне облачного провайдера» NIST**

Гибридное облако

## **83. Архитектура AWS (Amazon Web Services). Назначение и функции сервисов AWS.**

Службы AWS:

ü EC2 – Elastic Compute Cloud;

ü SQS – Simple QueueService;

ü EBS – Elastic block Service;

ü S3 – Simple Storage Service;

ü SNS – Simple Notification Service.

Назначение и функции сервисов AWS



# **Тема 12. Подходы к проектированию процессов выполняемых компонентами РС**

## **84. Клиенты распределенных ИС.**

Основное назначение:

* Предоставление пользователю возможности взаимодействовать с сервером и получать нужный вид обслуживания.
* В зависимости от природы (принципов, лежащих в основе) клиент-серверного взаимодействия, имеются различные способы реализации клиентского ПО.

**Сетевые пользовательские интерфейсы**

При работе в сети пользователю требуется доступ к ресурсам сервера. Имеется два основных пути достижения этой цели:

1. Обеспечить для работы с каждым видом ресурсов (видом сервиса) отдельную копию клиентского ПО. Например, файловый клиент, почтовый клиент и т.п. Вариант (а).

2. Использовать прямой доступ к удаленному сервису (серверу), за счет обеспечения только, подходящего пользовательского интерфейса (терминальный доступ), часто средствами локальной ОС. Это подход получил название **тонкий-клиент**. Внимание к нему не ослабевает, в связи с расширением популярности Интернет, а также использованием мобильных устройств. Вариант (b).

Пример: Система X-Window

Одним из старейших, но широко распространенных протоколов удаленного обеспечения пользовательского графического интерфейса является **X Window System** или просто **X**. В мире MS Windows аналогом является Terminal Service с использованием протокола RDP (Remote Desktop Protocol).

Является основой реализации графического UI в Unix и Unix-подобных ОС.

Организация X Window

Сердцем X является **X Kernel**. Оно содержит все драйвера устройств относящихся к терминалу (KVM – Keyboard, Video, Mouse). X Kernel обеспечивает низкоуровневый интерфейс управления экраном, а также перехватывает события связанные с работой клавиатуры и мыши. Это интерфейс поддерживается приложением, исполняющемся на сервере, библиотекой **Xlib**.

**X Kernel** и **X приложение** могут располагаться на разных машинах.

На сервере одновременно могут исполняться несколько приложений, взаимодействующих с X Kernel клиента. Управляет отображением приложение на X клиенте специальное приложение – Window manager. Особенностью X является то, что **X Kernel работает как сервер**, в то время как **X приложение играет роль клиента**.

**Тонкий сетевой клиент**

Недостатки X Window:

* X приложение в процессе своей работы посылает по сети команды, касающиеся управления дисплеем на X клиенте, которые последовательно выполняются X Kernel. Это синхронное поведение компонентов X в глобальных сетях, влечет за собой снижение производительности и увеличение задержек в работе X приложений.
* В идеале, X приложение должно разделять логику работы приложения (бизнес логику) и функции управления пользовательским интерфейсом, однако, часто это требование не выполняется, что также, влечет за собой замедление работы X клиентов и увеличение задержек в их работе.

Имеется несколько решений этих проблем X Window:

**Технология NX**: повсеместно известна как “NX”, разработана компанией NoMachine (2003). ;

**VNC**: Virtual Network Computing (конец 90-х). система удалённого доступа к рабочему столу компьютера, использующая [протокол](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%82%D0%B5%D0%B2%D0%BE%D0%B9_%D0%BF%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB) [RFB](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_RFB) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Remote FrameBuffer*, *удалённый* [*кадровый буфер*](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D0%B4%D1%80%D0%BE%D0%B2%D1%8B%D0%B9_%D0%B1%D1%83%D1%84%D0%B5%D1%80)).;

**THINC:** Thin-Client Internet Computing (2007). Архитектура системы тонкого клиента высокопроизводительных вычислений для локального и глобального окружений.

**Технология NX**

Основана на исходном протоколе X Window, со следующими улучшениями:

* Использование сжатия данных при передаче;
* Реализована передача только изменений, между смежными по времени образами.
* Использование протокола SSH для обеспечения шифрования соединения.

Является альтернативой X Window, в которой приложение полностью управляет отображением на удаленном дисплее, в плоть до пикселя.

Управление осуществляется путём передачи нажатий клавиш на клавиатуре и движений мыши с одного компьютера на другой и ретрансляции содержимого экрана через компьютерную сеть.

Система VNC платформонезависима: VNC-клиент, называемый VNC viewer, запущенный на одной операционной системе, может подключаться к VNC-серверу, работающему на любой другой ОС.

Существуют реализации клиентской и серверной части практически для всех операционных систем, в том числе и для [Java](https://ru.wikipedia.org/wiki/Java) (включая мобильную платформу J2ME).

К одному VNC-серверу одновременно могут подключаться множественные клиенты. Наиболее популярные способы использования VNC — удалённая техническая поддержка и доступ к рабочему компьютеру из дома.

**THINC**

Недостатком передачи сырых данных пикселей по сравнению с высоуровневыми протоколами такими как X Window является, то что невозможно использование семантики приложения.

В 2005г. Баррато предложил другой подход, который получил название THINC. Было предложено использовать ограниченное число высокоуровневых команд для управления отображением на дисплее клиента на уровне драйверов устройств.

Это более эффективно, чем прямое управление пикселями, но мене эффективно, чем обеспечивает X Window.

**Клиентское ПО прозрачного доступа**

Во многих случаях требуется обеспечить прозрачное выполнение обработки данных на клиенте и передачу результатов на сторону сервера, например передача штрих кодов, оплаченных наличными сумм и т.п. В этих случаях пользовательский интерфейс составляет небольшую часть клиентского ПО.

Прозрачный доступ обычно реализуется с помощью генерации клиентских заглушек на основе описаний интерфейса предоставляемого сервером.

Заглушка предоставляет такой же интерфейс, что и сервер, но она скрывает возможную разницу в архитектуре конкретных серверов, а также разницу в способах коммуникации с серверами.

В идеале клиент не должен знать где расположен сервер и даже тот факт, что он взаимодействует с удаленным сервером.

Имеются различные способы обеспечения прозрачности местоположения, миграции и перемещения сервера. Удобная схема именования является ключевым элементом прозрачности.

## **85. Сервера РС.**

**Общие проблемы создания серверов РС**

Сервер реализуется средствами исполнения процессов обеспечивающих реализацию сервиса, потребителями которого являются клиенты.

Имеется несколько вариантов построения сервера:

* Параллельная или интеративная обработка запросов.
* Реализация сервера с отслеживание состояние или без.

**Параллельный сервер против итерационного**

В случае использования параллельной работы сервера, поступающий запрос перехватывается соответствующим процессом приема запросов.

Для обработки запросов и формирования ответных сообщений используются отдельные потоки, запускаемые при получении очередного запроса.

После обработки запроса и отправки ответа, поток обработчик переходит в ожидание или закрывается.

Этот метод реализован в большинстве UNIX систем.

При интерактивной работе сервера серверный процесс сам реализует обработку запроса, а также отправку ответа клиенту.

**Прерываемый сервер**

При создании сервера следует принимать во внимание когда и как работа сервера может быть прервана. Существует несколько способов сделать это:

Один из подходов, кстати единственный, надежно работающий в современном Интернете (а иногда и единственно возможный), — пользователь немедленно закрывает клиентское приложение (что автоматически вызывает разрыв соединения с сервером), тут же перезапускает его и продолжает работу. Сервер, естественно, разрывает старое соединение, полагая, что клиент прервал работу.

Более правильный способ - разрабатывать клиент и сервер так, чтобы они могли пересылать друг другу сигнал *конца связи {out-ofband),* который должен обрабатываться сервером раньше всех прочих передаваемых клиентом данных. Здесь также возможны разные варианты:

Потребовать от сервера просматривать отдельную управляющую конечную точку, на которую клиент будет отправлять сигнал конца связи и одновременно (с более низким приоритетом) — конечную точку, через которую передаются нормальные данные (Пример FTP сервис).

Другой вариант — пересылать сигнал конца связи через то же соединение, через которое клиент пересылал свой запрос. В TCP, например, можно посылать срочные данные. Когда срочные данные достигают сервера, он прерывает свою работу (в UNIX- системах — по сигналу), после чего может просмотреть эти данные и обработать их.

**Сервер с фиксацией состояния**

Сервер с фиксацией состояния (stateful server) хранит и обрабатывает информацию о своих клиентах.

Типичным примером такого сервера является файловый сервер, позволяющий клиенту создавать локальные копии файлов, скажем, для повышения производительности операций обновления.

Подобный сервер поддерживает таблицу, содержащую записи пар (клиент, файл). Такая таблица позволяет серверу отслеживать, какой клиент с каким файлом работает и, таким образом, всегда определять самую «свежую» версию файла. Подобный подход повышает производительность операций чтения-записи, осуществляемых на клиенте.

**Достоинства и недостатки сервера с фиксацией состояния**

Достоинства: Рост производительности по сравнению с серверами без фиксации состояния, что является основной причиной разработки серверов с фиксацией состояния.

Недостатки: В случае сбоя сервера он вынужден восстанавливать свою таблицу с записями пар (клиент, файл), в противном случае не будет никакой гарантии в том, что работа происходит с последней обновленной версией файла. Как правило, серверы с фиксацией состояния нуждаются в восстановлении своего состояния в том виде, в котором оно было до сбоя.

**Сервер без состояния**

Сервер без фиксации состояния (stateless sewer) не сохраняет информацию о состоянии своих клиентов и может менять свое собственное состояние, не информируя об этом своих клиентов

Web-сервер, например, это сервер без фиксации состояния. Он просто отвечает на входящие HTTP-запросы, которые могут требовать загрузки файла как на сервер, так и (гораздо чаще) с сервера. После выполнения запроса web-сервер забывает о клиенте. Кроме того, набор файлов, которыми управляет web-сервер (возможно, в комбинации с файловым сервером), может быть изменен без уведомления клиентов об этом действии.

Достоинства: В случае архитектуры без фиксации состояния вообще нет необходимости принимать какие-то специальные меры по восстановлению серверов после сбоя. Они просто перезапускаются и работают, ожидая запросов клиента.

Недостатки: Отсутствие возможности отслеживания состояния сессии клиента.

**Серверы объектов**

Сервер объектов (object server) — это сервер, ориентированный на поддержку распределенных объектов.

Важная разница между стандартным сервером объектов и другими (более традиционными) серверами состоит в том, что сам по себе сервер объектов не предоставляет конкретной службы.

Конкретные службы реализуются объектами, расположенными на сервере. Сервер предоставляет только средства обращения к локальным объектам на основе запросов от удаленных клиентов. Таким образом, можно относительно легко изменить набор служб, просто добавляя или удаляя объекты.

## **86. Кластера серверов. Глобальные кластеры серверов.**

В вычислительных кластерах один узел играет роль управляющего. Остальные узлы являются вычислительными.

Управляющий кластер исполняет ПО промежуточного слоя, обеспечивающее управление кластером и выполнение вычислительными узлами параллельных фрагментов срвместно решаемой задачи.

Вычислительные кластера могут быть как локальными так и глобальными.

Кластер серверов представляет собой распределенную систему имеющую логическую в 3-х уровневую структуру:

1-й уровень – логический коммутатор, распределяющий запросы клиентов между серверами кластера;

2-й уровень – сервера приложений/вычислений, обрабатывающие запросы клиентов;

3-й уровень – сервера обработки данных.

Когда кластер предлагает выполнение нескольких сервисов, они могут быть распределены на различных машинах кластера.

Как результат, некоторые машины могут находиться в режиме ожидания запросов, в то время как другие могут испытывать перегрузки.

Решением может быть – объединение машин в кластер высокой готовности и использование виртуальных машин.

**Глобальные кластеры серверов**

Облачные провайдеры Amazon и Google имеют собственные ЦОД в разных частях мира, в которых размещаются сервера доступа к облачным сервисам.

В этих ЦОД размещаются виртуальные машины на базе которых поставщики облачных услуг предоставляют возможность пользователям создавать свои собственные глобальные распределенные системы, содержащие большое количество виртуальных машин объеденных сетью поверх Интернет.

*Важнейшей характеристикой такой системы является место расположения серверов служб, которые должны располагаться по близости с потенциальными пользователями системы.*

*Если близость расположения серверов к местам расположения пользователей не является важной, то можно разместить все ВМ в одном ЦОД и получить преимущество в производительности за счет скорости исполнения межпроцессных коммуникаций по локальной сети ЦОД с низкими величинами задержек.*

*Если близость пользователей к серверам кластера важна, то важность приобретает и способ диспетчирования запросов, так как они должны направляться ближайшему серверу кластера.*

**Политика перенаправления**

Необходимость выбора сервера ближайшего к источнику запроса порождает проблему политики перенаправления.

Если предположить, что по аналогии с локальными кластерами серверов диспетчирование будет осуществляться с помощью коммутатора запросов, то диспетчер должен иметь способность оценивать величину задержки передачи между клиентом и различными серверами.

Как только сервер будет выбран, диспетчер должен уведомить об этом клиента и перенаправить его запрос серверу. Для этого могут использоваться различные механизмы:

Использование в качестве диспетчера DNS сервер; Клиент делает запрос на разрешение имени сервера и ему возвращается адрес ближайшего к нему сервера кластера (конечно диспетчер должен знать IP адрес клиента).

## **87. Миграция кода.**

**Основания для переноса кода**

* Традиционно перенос кода в распределенных системах происходит в форме переноса процессов (process migration)у в случае которых процесс целиком переносится с одной машины на другую.
* Поддержка переноса кода может также помочь повысить производительность на основе параллелизма, но без обычных сложностей, связанных с параллельным программированием. Типичным примером может быть поиск информации в Web.
* Относительно несложно реализовать поисковый запрос в виде небольшой мобильной программы, переносимой с сайта на сайт. Создав несколько копий этой программы и разослав их по разным сайтам, мы можем добиться линейного возрастания скорости поиска по сравнению с единственным экземпляром программы.
* Помимо повышения производительности существуют и другие причины поддержания переноса кода. Наиболее важная из них — это гибкость.

**Модели переноса кода**

Перенос кода в широком смысле связан с переносом программ с машины на машину с целью исполнения этих программ в нужном месте.

Для лучшего понимания различных моделей переноса кода используем шаблон, состоящий из:

* *Сегмент кода — это часть, содержащая набор инструкций, которые выполняются* в ходе исполнения программы.
* *Сегмент ресурсов содержит ссылки на* внешние ресурсы, необходимые процессу, такие как файлы, принтеры, устройства, другие процессы и т. п.
* *Сегмент исполнения используется для хранения* текущего состояния процесса, включая закрытые данные, стек и счетчик программы.

Различают 2 модели переноса кода, характеризующиеся степенью мобильности:

* модель слабой мобильности (weak mobility). Согласно этой модели допускается перенос только сегмента кода, возможно вместе с некоторыми данными инициализации.
* Модель сильной мобильности (strong mobility). В этом случае переносится также и сегмент исполнения.

**Инициатор переносимости кода**

Независимо от того, является мобильность слабой или сильной, следует провести разделение на:

* системы с переносом, инициированным отправителем,
* системы с переносом, инициированным получателем.

При переносе, инициированном отправителем, перенос инициируется машиной, на которой переносимый код постоянно размещен или выполняется.

Обычно перенос, инициированный отправителем, происходит при загрузке программ на вычислительный сервер.

При переносе, инициированном получателем, инициатива в переносе кода принадлежит машине-получателю. Пример такого подхода — Java-апплеты.

**Четыре варианта переноса кода**

Различные варианты переноса кода в зависимости от:

* вида мобильности (сильная/слабая);
* типа инициатора (отправитель/полу-чатель);

порождают следующие парадигмы (модели):

Ø Клиент-сервер (CS) (нет мобильности кода);

Ø Удаленные вычисления (REV) (слабая/отправителем);

Ø Код по требованию (CoD)

(слабая/получателем);

Ø Мобильный агент (MA) (сильная/отправителем).

**Влияние миграции кода на время ответа ВМ**

При миграции виртуальной машины время неактивности этой ВМ составляет всего несколько минут.

# **Тема 13. Организация коммуникаций в распределенных ИС.**

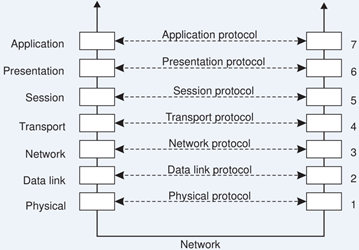
## **88. Роль связи в распределенных системах. Протоколы промежуточного уровня. Коммуникаций на уровне ПО промежуточного слоя. Критерии коммуникаций. Роль связи в распределенных системах**

Связь между процессами, исполняемыми на взаимодействующих друг с другом узлах РИС – определяет суть распределенных систем.

Обмен информацией между узлами выполняется с использованием протоколов сетевой модели, используемой в распределенной системе.

Сетевая модель относится к инфраструктуре, в которой функционирует РС.

**Сетевая модель OSI/ISO**



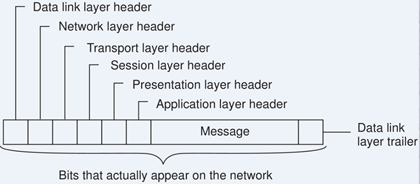
Уровни, интерфейсы и протоколы модели OSI

Протоколы бывают:

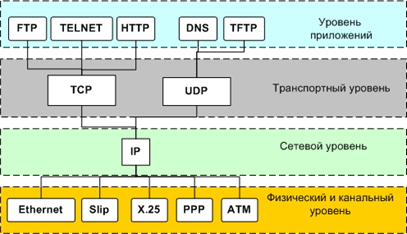
- с установлением соединения;

- без установления соединения.

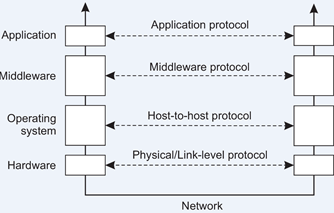
**Структура сетевого сообщения**

****

В настоящее время стек протоколов TCP/IP является основной сетевой моделью, поддерживаемой всеми операционными системами узлов распределенных систем.



**Протоколы промежуточного уровня**

****

К промежуточному уровню относятся приложения, логически помещаемые на прикладной уровень, но содержащие множество протоколов общего назначения.

Примеры протоколов поддерживаемых ПО промежуточного уровня:

- протокол DNS;

- протоколы аутентификации;

-- протоколы распределенного подтверждения;

- протоколы распределенной блокировки;

-- протоколы удаленного вызова процедур;

- удаленного вызова методов; удаленного обращения к объектам;

-- протоколы обмена сообщениями.

По сравнению с моделью OSI сеансовый уровень и уровень представления заменены одним промежуточным уровнем, который содержит не зависящие от приложений протоколы, эти протоколы нельзя поместить на более низкие уровни. Транспортные службы также могут быть представлены в виде служб промежуточного уровня.

**КОММУНИКАЦИЙ НА УРОВНЕ ПО ПРОМЕЖУТОЧНОГО СЛОЯ. КРИТЕРИИ КОММУНИКАЦИЙ**

Для понимания альтернатив организации связи на уровне ПО промежуточного слоя необходимо определить основные критерии таких коммуникаций.

Такими критериями, коммуникаций являются:

1.Время хранения сообщения.

2.Синхронность выполнения, различают типы .

3.Особенность природы данных.

## **89. Виды связи. Сохранная и нерезидентная связь. Синхронная и асинхронная связь. Дискретные и потоковые данные Сохранная и нерезидентная связь**

Время хранения сообщения, по этому критерию различают типы:

* Сохранная (persistent) связь (коммуникации) - сообщение, предназначенное для отсылки, хранится в коммуникационной системе до тех пор, пока его не удастся передать получателю;
* Нерезидентная (transient - краткосрочная) связь (коммуникации) - сообщение хранится в системе только в течение времени работы приложений, которые отправляют и принимают это сообщение;

**Синхронная и асинхронная связь**

Синхронность выполнения, различают типы:

* Асинхронная связь - является немедленное после отправки письма продолжение работы отправителя. Это означает, что письмо сохраняется в локальном буфере передающего хоста или на ближайшем коммуникационном сервере;
* Синхронная связь - *отправитель блокируется до того момента* пока его сообщение не будет сохранено в локальном буфере принимающего хоста или доставлено реальному получателю. Наиболее жесткая форма синхронного взаимодействия предполагает, что отправитель остается блокированным и на время обработки его сообщения получателем.;

**Дискретные и потоковые данные**

Особенность данных:

* Отдельные блоки данных (дискретные данные);
* Потоки данных (непрерывные последовательности данных).

## **90. Обобщенная организация коммуникационной системы. Организация коммуникаций в РИС на примере службы электронной почты. Типы коммуникаций, используемые в службе электронной почты. Обобщенная организация коммуникационной системы, хосты которой соединяются через сеть**

Приложения всегда выполняются на хостах, а каждый хост предоставляет интерфейс с коммуникационной системой, через который сообщения могут передаваться.

Хосты соединены сетью коммуникационных серверов, которые отвечают за передачу (или маршрутизацию) сообщения между хостами.

Без потери общности можно предположить, что каждый из хостов связан только с одним коммуникационным сервером.

**Пример: электронная почта**

Для понимания различных альтернатив в коммуникациях промежуточного уровня необходимо рассмотреть модель взаимодействия клиент-сервер, на примере **электронной почты.**

Компоненты службы эл. Почты:

* МТА – транспортный агент, сервер доставки эл.почты (протокол SMTP)
* ДА – агент доставки эл. почты (утилита mail) доставляет сообщение в почтовый ящик пользователя;
* UA – пользовательский агент, доставляет сообщение на машину пользователя (протокол POP3/IMAP4);
* Клиент SMTP – клиентское ПО отправки сообщений эл. почты (SMTP);

**Типы коммуникаций, используемые в службе электронной почты**

Система электронной почты — это типичный пример *сохранной связи.*

Сообщение, отправляемое клиентом на сервер, сохраняется на сервере для дальнейшей доставки с помощью средств ПО промежуточного слоя, т.е. имеет место **сохранность коммуникации**.

**Асинхронность** коммуникации имеет место, когда сервер начинает доставку сразу же после приема сообщения, а клиент, может продолжить свою работу после приема сообщения сервером.

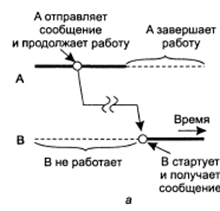
**Синхронность** предполагает ожидание клиентом завершения выполнения сервером обработки запроса.

## **91. Сохранная синхронная и асинхронная связь.**

**сохранная асинхронная связь**

В случае сохранной асинхронной связи сообщение сохраняется в буфере либо локального хоста, либо первого коммуникационного сервера.

Этот вид связи обычно используется в системах электронной почты.



**сохранная синхронная связь**

В случае сохранной синхронной связи сообщения хранятся только на принимающем хосте.

Отправитель блокируется до момента сохранения сообщения в буфере получателя. Отметим, что приложение, принявшее сообщение, не обязано сохранять его на своем локальном хосте.

«Усеченный» вариант сохранной синхронной связи состоит в том, что отправитель блокируется до момента сохранения сообщения на коммуникационном сервере, соединенном с принимающим хостом.



## **92. Нерезидентная синхронная и асинхронная связь.**

**нерезидентная асинхронная связь.**

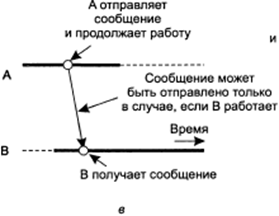
Нерезидентная асинхронная связь характерна для служб дейтаграмм транспортного уровня, таких как UDP.

Когда приложение отправляет сообщение, оно временно сохраняется в локальном буфере передающего хоста, после чего отправитель немедленно продолжает работу.

Параллельно коммуникационная система направляет сообщение в точку, из которой, как ожидается, оно сможет достигнуть места назначения, возможно, с сохранением в локальном буфере.

Если получатель в момент прихода сообщения на принимающий хост этого получателя неактивен, передача обрывается.

Другой пример нерезидентной асинхронной связи — асинхронный вызов RPC. При асинхронных вызовах RFC клиент синхронизируется с сервером, ожидая, пока его запрос будет принят на дальнейшую обработку.



**нерезидентная синхронная связь с синхронизацией по приему**

Нерезидентная синхронная связь существует в различных вариантах.

В наиболее слабой форме, основанной на подтверждениях приема сообщений, отправитель блокируется до тех пор, пока сообщение не окажется в локальном буфере принимающего хоста.

После получения подтверждения отправитель продолжает свою работу.

****

**нерезидентная синхронная связь с синхронизацией по доставке**

В ориентированной на доставку нерезидентной синхронной связи отправитель блокируется до тех пор, пока сообщение не будет доставлено получателю для дальнейшей обработки.

При синхронных вызовах RРC клиент синхронируется с сервером, ожидая, пока его запрос будет обработан сервером.

****

**нерезидентная синхронная связь с синхронизацией по ответу**

Наиболее жесткая форма — ориентированная на ответ нерезидентная синхронная связь — предполагает блокировку отправителя до получения ответного сообщения с другой стороны, как в поведении запрос-ответ при взаимодействии клиент-сервер.

Эта схема характерна также для механизмов RPC и RMI.



## **93. Применение различных типов коммуникаций в РС. Дискретные и потоковые коммуникации.**

**Применение различных типов коммуникаций в РС**

* Асинхронная связь – является более быстрой, более гибкой, но более сложной в программной реализации, т.к. ответ может поступить в непредсказуемый момент времени. Используется в системах, основанных на событиях.
* Полностью синхронный вариант коммуникаций может сильно замедлить процесс обработки, но программная реализация его более проста и понятна.
* При использовании мультипотоковых программных реализаций ПО промежуточного слоя блокировка одного из процессов, не представляет собой большой проблемы, т. к. отдельный поток может отслеживать (ожидать) появление ответных сообщений.

**Дискретные и потоковые коммуникации**

* **Дискретными** – называются коммуникации в рамках которых передаются отдельные блоки данных, порядок передачи которых может быть изменен. А задержка и стабильность скорости передачи не имеют значения для передаваемых блоков данных.
* **Потоковыми** – называются данные, передаваемые в одном направлении; сессия содержит множество сообщений, поступающих от отправителя в определенном порядке, который определяется либо порядком передачи, либо порядком передачи по времени, либо еще чем-либо.

# **Тема 14. Основные виды коммуникаций распределенных систем.**

## **94. Удаленный вызов процедур. RPC и модель Клиент-Сервер. Реализация прозрачности: Использование программных заглушке (Stubs). Передача значений параметров. Согласование интерфейсов. Синхронный и асинхронный RPC. Проблемы RPC. Различные реализации RPC. RMI.**

**Мотивация использования RPC**

* Использовать низкоуровневый обмен сообщениями на основе функций, встроенных в язык программирования (примитивов) send и receive. primitives.
* Стремление к обеспечению прозрачности на уровне доступа.
* Сокрытие различий в представлениях данных на машинах, в понимании процессов обмена сообщениями и т.п. Другими словами - сделать RPC прозрачным: вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.
* Упрощение программирования процесса обмена информацией за счет использования методов подобных тем, которые применяются при обмене через разделяемую память при локальном вызове процедур.

**Модель Remote Procedure Call (RPC)**  
Использование высокоуровневых интерфейсов сетевых коммуникаций, предоставляемых ОС. Основан на модели вызова локальных процедур в рамках одного процесса.

Запрос клиента: оформляется как вызов процедуры выполнения функции на сервере. Ответ сервера: оформляется как возвращение результата выполнения вызванной функции.

**Передача параметров при локальном вызове процедуры**

* Стек до вызова функции read.
* Стек во время работы вызванной процедуры

Приложения RPC похожи на другие структурированные приложения:

у них есть основная программа, которая для выполнения специфических задач вызывает процедуры или библиотеки процедур.

Отличие приложений RPC от обычных программ в том, что некоторые библиотеки процедур в приложениях RPC выполняются на удаленных компьютерах, а некоторые — на локальном.

Для приложения RPC все процедуры кажутся локальными.

Функционирует приложение RPC следующим образом. В процессе своей работы оно вызывает как локальные процедуры, так и процедуры, отсутствующие на локальной машине.

Для обработки последнего случая приложение связывается с локальной DLL, которая содержит интерфейсные процедуры (stub procedures) для всех удаленных процедур. В простой программе интерфейсные процедуры статически связываются с приложением, но в компоненте большего размера они включаются в отдельные DLL.

В DCOM обычно применяется последний метод. Интерфейсная процедура имеет то же имя и тот же интерфейс, что и удаленная процедура, но вместо выполнения соответствующей операции она просто преобразует переданные ей параметры для передачи по сети — такой процесс называется маршалингом.

Маршалинг заключается в упорядочении параметров и их упаковке в определенном формате.

Далее интерфейсная процедура вызывает процедуры библиотеки RPC периода выполнения, и они находят компьютер, на котором расположены удаленные процедуры, определяют используемые этим компьютером механизмы транспорта и посылают запрос при помощи локального программного обеспечения сетевого транспорта. Когда удаленный сервер получает запрос RPC, он выполняет обратное преобразование параметров

Локальный вызов процедур. Выполнение

Управление передается вызываемой функции (процедуре)

Вызванная функция выполняется и по завершении работы возвращает результаты либо через параметры (стек), либо сохраняет их в регистрах CPU.

Выполняется извлечение данных из стека.

Процесс, вызывавший процедуру, продолжает свою работу.

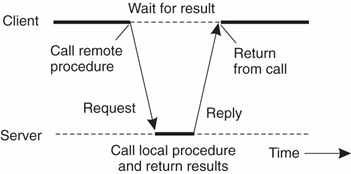
**Особенности Remote Procedure Calls**

Основные операции RPC выполняются параллельно с выполнением процесса вызвавшем RPC процессом.

Процесс, вызвавший RPC, выполняет удаленный вызов при этом его работа приостанавливается до завершения выполнения вызываемой функции и возвращения результатов.

Параметры передаются на машину, на которой процедура будет исполняться.

Когда выполнение вызываемой процедуры будет завершено, результаты передаются назад на машину источник вызова, после чего клиентский процесс продолжит свою работу.



**RPC и модель Клиент-Сервер**

RPC лежит в основе большинства систем клиент-сервер.

Клиенты формулируют запросы к серверам в виде вызовов процедур.

Прозрачность доступа обеспечивает реализация механизма RPC

## **95. Коммуникации, ориентированные на передачу сообщений (Message Oriented Communication). Сокеты. Использование сокетов клиентом и сервером. Коммуникации на основе сокетов. Общая схема взаимодействия клиента и сервера с использованием сокетов.**

Несмотря на то, что RPC и RMI поддерживают прозрачность доступа, однако не всегда это делается походящим способом.

Коммуникации ориентированные на обмен сообщениями обладают большей гибкостью. Эти коммуникации основываются на уровне транспортных протоколов – транспортный уровень OSI.

Стандартизованными интерфейсами к транспортному уровню OSI являются Сокеты:

* (Berkeley UNIX);
* XTI (X/Open Transport Interface), ранее известные как TLI (AT&T model).

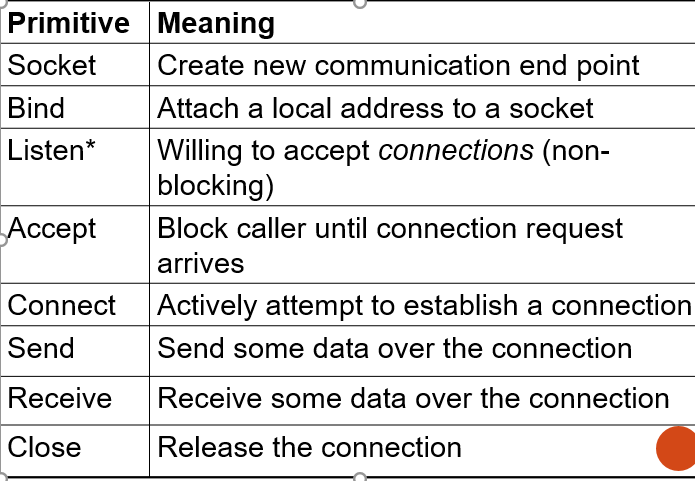
**Сокеты**

Связь между конечными точками используется приложениями для записи/чтения в/из сети.

Механизм сокетов предоставляет набор базовых примитивов (функций) для выполнения операций по передаче и приему данных

Сокеты являются абстракцией реальных коммуникаций с конечными точками обслуживания средствами локальной ОС.

Адрес сокета: Адрес IP + номер порта



**Коммуникации на основе сокетов**

Используя сокеты, клиенты и серверы могут устанавливать между собой связь в рамках сессии, ориентированной на соединение.

На серверах используются первые четыре функции (socket, bind, listen, accept), в то время как на клиентах используются примитивы (функции) – socket и connect.

Затем на клиенте и сервере поочередно выполняются операции записи-чтения (client/write, server/read, server/write, client/read) а вслучае закрытия соединения операция close, закрывающая соединение.

**Общая схема взаимодействия клиента и сервера с использованием сокетов. Сервер.**

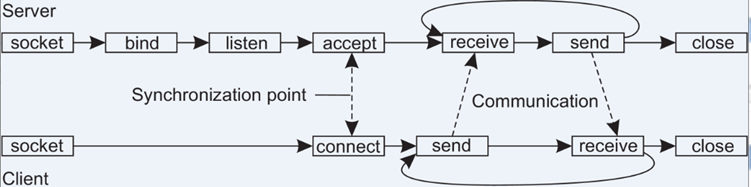
Общая схема взаимодействия клиента и сервера с использованием сокетов для коммуникаций, ориентированных на соединение:

Сервер:

Примитив bind выполняет привязку локального адреса к только что созданному сокету. Например, сервер должен связать IP-адрес своей машины с номером порта (возможно, общеизвестным) сокета. Привязка сообщает операционной системе, что сервер намерен получать сообщения только на указанные адрес и порт.

Примитив listen применяется только для коммуникаций, ориентированных на соединение. Это неблокирующий вызов, требующий от локальной операционной системы зарезервировать буфер для определенного максимального количества соединений, которое вызывающий процесс намерен поддерживать.

Вызов примитива accept блокирует вызывающий процесс до прихода запроса на соединение. Когда этот запрос придет, локальная операционная система создаст новый сокет с теми же свойствами, что и у базового, и возвратит его вызывающему процессу.



**Общая схема взаимодействия клиента и сервера с использованием сокетов. Клиент.**

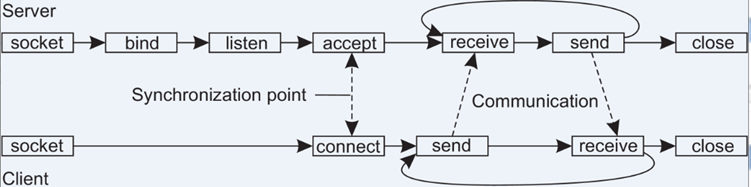
На стороне клиента все начинается с создания сокета при помощи примитива socket, однако в явной привязке сокета к локальному адресу нет необходимости, поскольку операционная система может динамически выделить порт при установлении соединения.

Примитив connect требует, чтобы вызывающий процесс указал адрес транспортного уровня, на который будет отправлен запрос на соединение.

Клиент блокируется до тех пор, пока соединение не будет установлено. После установления соединения

стороны начинают обмениваться информацией при помощи примитивов write и read, предназначенных для посылки и приема данных соответственно.

Наконец, закрытие соединения при использовании сокетов симметрично и может быть осуществлено как клиентом, так и сервером путем вызова примитива close.



## **96. Интерфейс передачи (обмена) сообщений (MPI-Message-Passing Interface). Коммуникации в MPI. Некоторые из наиболее используемых примитивов MPI. Асинхронная и синхронная связь в MPI. Причина поддержки MPI большого числа вариантов взаимодействия. Приложения MPI против Клиент-сервер.**

Сокеты предоставляют низкоуровневый интерфейс доступа к глобальным (TCP/IP-based) сетям.

Распределенные системы, работающие на высокоскоростных сетях в высокопроизводительных кластерных системах требуют более эффективных протоколов обмена сообщениями, обеспечивающих эффективные варианты буферизации и синхронизации.

Часто высокопроизводительные мультикомпьютерные вычислительные системы имеют свои собственные коммуникационные библиотеки.

Необходимость в платформе обмена сообщениями независящей от аппаратной платформы привела к необходимости создания стандарта обмена сообщениями – МРI.

MPI – это библиотека спецификаций обмена сообщениями, предложенная в качестве стандарта комитетом производителей, реализаторов и пользователей.

MPI разработан для обеспечения обмена сообщениями между узлами вычислительного кластера на основе асинхронной сохранной связи сообщений при выполнении параллельных вычислений. Но поддерживаются и синхронные взаимодействия.

MPI используется во многих системных окружениях, включая как вычислительные кластеры, так и гетерогенные сети.

MPI является платформо-независимым решением.

MPICH2 – это популярная реализация стандарта MPI.

**Коммуникации в MPI**

MPI предполагает использование базовых сетей и не предусматривает ничего, напоминающего коммуникационные серверы. Кроме того, он предусматривает, что серьезные сбои в системе, такие как аварии процессов или участков сети, фатальны и не могут быть восстановлены автоматически.

Коммуникации предполагаются между группой процессов известных друг другу.

Для группу прцессов назначается свой уникальный идентификатор - groupID. Для каждого процесса, включенного в группу, назначается свой идентификатор процесса – processID.

Пара (groupID, processID) выполняют роль адреса процесса при коммуникациях на основе MPI.

**Операторы (примитивы) MPI передачи сообщений. Асинхронная связь.**

* Оператор MPI\_bsend: обеспечивает асинхронную сохранную связь между процессами. После выполнения этой операции, передающий процесс (sender) продолжает выполнение кода, одновременно с копирование сообщения в локальный буфер для последующей доставки процессу получателю (bsend = buffer send). Сообщение будет скопировано в буфер на принимающей машине и позднее будет получено процессом в ответ на выдачу оператора (примитива) receive.

**Операторы (примитивы) MPI передачи сообщений. Синхронная связь**

* MPI\_send: блокирует передачу до тех пор пока сообщение не будет скопировано в локальный или удаленный буфер. Семантика зависит от реализации.
* MPI\_ssend: передающий процесс блокируется до тех пор пока запрос не будет принят приемником.
* MPI\_sendrecv: отправляет сообщение и ожидает ответа (по сути аналогично RPC)

В целом MPI поддерживает большое число функций (более 100), предназначенных для организации сетевых взаимодействий в многомашинных вычислительных системах (вычислительных кластерах).

**Операторы (примитивы) MPI.Прием сообщений.**

Операция MPI\_recv вызывается для приема сообщения и блокирует запустивший процесс до прихода сообщения.

Существует также и асинхронный вариант этой операции под именем MPI\_irecv, вызовом которого получатель показывает, что он готов к приему сообщений. Получатель может проверить, имеются ли пришедшие сообщения, или заблокироваться в ожидании таковых.

Причина поддержки такого разнообразия вариантов взаимодействия состоит в том, что разработчики систем MPI должны иметь все возможности для оптимизации производительности многомашинных вычислительных систем (MIMD – системы по классификации Флина).

Семантика коммуникационных примитивов MPI не всегда проста, и иногда замена различных примитивов никак не влияет на правильность программы.

**Приложения MPI против Клиент-сервер**

Процессы в параллельных системах на базе MPI действуют более независимо, подобно пирам в одноранговых системах.

Коммуникации могут обеспечивать обмен сообщениями в нескольких направлениях.

Коммуникации в системах клиент сервер являются более структурированными.

## **97. Промежуточный уровень ориентированный на сообщения. Сохранная связь на основе сообщений. Системы очередей сообщений. Модель очередей сообщений. Варианты слабосвязанных взаимодействий с использованием очередей.**

Системы очередей сообщений создают расширенную поддержку асинхронной сохранной связи.

Смысл этих систем заключается в том, что они предоставляют возможность промежуточного хранения сообщений, не требуя активности во время передачи сообщений ни от отправителя, ни от получателя.

Их существенное отличие от сокетов Беркли и интерфейса MPI состоит в том, что системы очередей сообщений обычно предназначены для поддержки обмена сообщениями, занимающего минуты, а не секунды или миллисекунды.

**Модель очередей сообщений**

Основная идея, лежащая в основе систем очередей сообщений, состоит в том, что приложения общаются между собой путем помещения сообщений в специальные очереди.

Эти сообщения передаются по цепочке коммуникационных серверов и в конце концов достигают места назначения, даже в том случае, если получатель в момент отправки сообщения был неактивен.

На практике большинство коммуникационных серверов напрямую соединены друг с другом. Другими словами, сообщение обычно пересылается непосредственно на сервер получателя.

В принципе каждое приложение имеет собственную очередь, в которую могут посылать сообщения другие приложения. Очередь может быть прочитана только связанным с ней приложением, при этом несколько приложений могут совместно использовать одну очередь.

Важный момент в системах очередей сообщений СОСТОИТ В ТОМ, ЧТО отправитель обычно в состоянии гарантировать только попадание сообщения — рано или поздно — в очередь получателя. Никакие гарантии относительно того, будет ли сообщение действительно прочитано, невозможны, это полностью определяется поведением получателя.

Подобная семантика определяет слабосвязанное взаимодействие. Именно поэтому у получателя нет необходимости быть активным в то время, когда сообщение пересылается в его очередь.

**Варианты слабосвязанных взаимодействий с использованием очередей**

Отправитель и получатель могут выполняться абсолютно независимо друг от друга.

На самом деле, как только сообщение поставлено в очередь, оно будет оставаться в ней до удаления, независимо от того, активен его отправитель или его получатель.

Возможны следующие варианты:

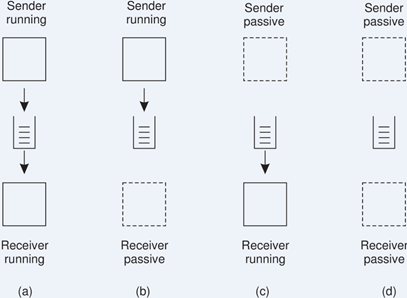
a) отправитель и получатель в ходе всего процесса передачи сообщения находятся в активном состоянии.

b) активен только отправитель, в то время как получатель отключен, то есть находится в состоянии, исключающем возможность доставки сообщения. Тем не менее отправитель все же в состоянии отправлять сообщения.

c) комбинация из активного получателя

и пассивного отправителя. В этом случае получатель может прочитать сообщения, которые были посланы ему ранее, наличия работающих отправителей этих сообщений при этом совершенно не требуется.

d) вариант когда система сохраняет и передает сообщения, даже при неработающих отправителе и получателе.



## **98. Базовый интерфейс очереди. Функции базового интерфейса очереди. Общая архитектура системы очередей сообщений. Управление очередями в распределенных системах. Ретрансляторы очередей. Маршрутизаторы очередей. Брокеры сообщений.**

Сообщения в принципе могут содержать любые данные. Единственно важный момент — они должны быть правильно адресованы.

На практике адресация осуществляется путем предоставления уникального в пределах системы имени очереди, в которую направляется письмо.

В некоторых случаях размер сообщений может быть ограничен, несмотря на то, что, возможно, базовая система в состоянии разбивать большие сообщения на части и собирать их обратно в единое целое абсолютно прозрачно для приложений.

Эффект подобного подхода — В ТОМ, что базовый интерфейс, предоставляемый приложениям, в результате можно сделать очень простым.

Примитив put вызывается отправителем для передачи сообщения базовой системе, где оно будет помещено в соответствующую очередь. Это неблокирующий вызов.

Примитив get — это блокирующий вызов, посредством которого авторизованный процесс может извлечь самое старое сообщение из определенной очереди. Процесс блокируется только в том случае, если очередь пуста. Варианты этого вызова включают в себя поиск в очереди определенного сообщения, например, с использованием приоритетов или образца для сравнения.

Неблокирующий вариант представлен примитивом poll. Если очередь пуста или искомое сообщение не найдено, вызвавший этот примитив процесс продолжает свою работу.

И, наконец, большинство систем очередей сообщений поддерживают также процесс вставки дескриптора функции обратного вызова (callback function), которая автоматически вызывается при попадании сообщения в очередь (примитив notify).

Обратные вызовы могут быть также использованы для автоматического запуска процесса, который будет забирать сообщения из очереди, если ни один процесс в настоящее время не запущен. Подобное поведение обычно реализуется при помощи демона на стороне получателя, который постоянно проверяет очередь на наличие входящих сообщений и поступает в соответствии с результатами проверки.

Общая архитектура системы очередей сообщений

Важно понимать, что набор очередей разнесен по множеству машин.

Соответственно, для того чтобы система очередей сообщений могла перемещать сообщения, она должна поддерживать отображение очередей на сетевые адреса.

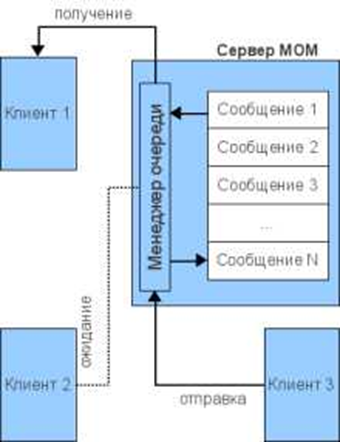
На практике это означает, что она должна поддерживать базу данных (возможно, распределенную) имен очередей (queue names) в соответствии с их сетевым местоположением, как показано на рис.

Это отображение полностью аналогично использованию системы доменных имен (DNS) для электронной почты Интернета. Так, например, для отсылки почты на логический почтовый адрес steen@cs.vu.nl почтовая система запрашивает у DNS сетевой адрес (то есть IP-адрес) почтового сервера получателя, чтобы затем использовать его для передачи сообщений.

**Управление очередями в распределенных системах**

Очереди управляются менеджерами очередей (queue managers). Обычно менеджер очередей взаимодействует непосредственно с отправляющими и принимающими сообщения приложениями.

Существуют, однако, и специализированные менеджеры очередей, которые работают как: маршрутизаторы, или ретрансляторы.



**Ретрансляторы очередей**

Ретрансляторы перенаправляют приходящие сообщения другим менеджерам очередей.

Ретрансляторы могут быть удобны по нескольким причинам. Так, во многих системах очередей сообщений не существует общих служб именования, которые могли бы динамически поддерживать отображение имен на адреса. Вместо этого топология сети с очередями сделана статической, а каждый менеджер очередей имеет копию отображения очередей на адреса.

В крупномасштабных системах очередей сообщений подобный подход легко может привести к трудностям в управлении сетью.

Другой причиной использования ретрансляторов является их способность производить вторичную обработку сообщений. Так, например, в целях безопасности или защиты от сбоев может быть необходимо ведение журнала сообщений.

Специальный тип ретрансляторов, в состоянии работать как шлюз, преобразуя сообщения в удобный для получателя вид.

И, наконец, ретрансляторы могут использоваться для групповой рассылки. В этом случае входящее сообщение просто помещается в каждую из исходящих очередей.

**Маршрутизаторы очередей**

Когда отправитель А помещает в локальную очередь сообщение для получателя В, это сообщение передается на ближайший маршрутизатор R1.

При этом маршрутизатор знает, что делать с этим сообщением, и передает его в направлении В. Так, R1 может понять по имени 5, что сообщение следует передать на маршрутизатор R2.

Таким образом, при добавлении или удалении очередей обновление информации потребуется только маршрутизаторам, в то время как остальным менеджерам очередей достаточно будет знать только местоположение ближайшего маршрутизатора.

Следовательно, маршрутизаторы могут помочь в создании масштабируемых систем очередей сообщений.

Однако по мере роста сети очередей сообщений становится ясно, что ручное конфигурирование невозможно при больших размерах сети.

Единственным решением будет использование динамической схемы маршрутизации, такой же как в компьютерных сетях.

**Брокеры сообщений**

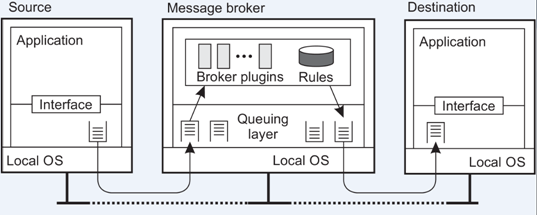
Важнейшей областью применения систем очередей сообщений является интеграция существующих и новых приложений в единые согласованные распределенные информационные системы.

Интеграция требует, чтобы приложения понимали сообщения, которые они получают, т.е. управляемые сообщения должны иметь тот формат, которого ожидает получатель.

Альтернатива состоит в том, чтобы принять единый формат сообщений, как это было сделано с традиционными сетевыми протоколами.

К сожалению, этот подход для систем очередей сообщений в общем случае неприменим. Проблема состоит в уровне абстракций, которыми оперируют эти системы. Единый формат сообщений имеет смысл, только если набор процессов, использующих этот формат, в реальности достаточно однотипен.

В системах очередей сообщений преобразование производится на специальных узлах сети массового обслуживания, известных под названием брокеров сообщений.



Брокер сообщений работает как шлюз прикладного уровня в системе очередей сообщений.

Его основная задача — преобразование входящих сообщений в формат, который понимается целевым приложением.

Для системы очередей сообщений брокер сообщений — это просто еще одно приложение

## **99. Сервисы MOM. Виды сервисов МОМ. API и протоколы систем очередей сообщений. API с поддержкой Message Oriented Middleware (MOM).**

Сервисы MOM хорошо зарекомендовали себя в сильно распределенных приложениях, используемых в гетерогенной сети с медленными и ненадежными соединениями. Это, во-многом, достигается благодаря поддержке уровней «качества обслуживания»:

* надежная доставка сообщений — система MOM гарантирует, что в процессе обмена ни одно сообщение не будет утеряно;
* гарантированная доставка сообщений — сообщение доставляется адресату немедленно или через заданный промежуток времени, не превышающий определенного значения (в случае, если сеть в данный момент не доступна);
* застрахованная доставка сообщений — каждое сообщение доставляется только один раз.

**Виды сервисов МОМ**

Очереди сообщений представляют собой мощный, гибкий и в то же время простой механизм межпрограммного взаимодействия.

Помимо приведенной, можно сказать классической, схемы с очередями, разработаны и используются сервисы MOM с непосредственной передачей сообщений и на основе подписки.

* Системы с непосредственной передачей сообщений используют логическое сетевое соединение для обмена сообщениями между взаимодействующими приложениями. Эта схема удобна в тех случаях, когда клиенты и серверы сообщений используются в сильно связанной сетевой инфраструктуре и синхронизированы по времени.
* Сервисы MOM, обслуживающие клиентов по подписке/публикации работают по принципу, напоминающему почтовую рассылку: одно приложение публикует информацию в сети, а другие подписываются на эту публикацию для получения необходимых данных. Взаимодействующие таким способом приложения полностью независимы друг от друга, что представляет возможности динамической реконфигурации всей распределенной системы.

**API и протоколы систем очередей сообщений**

Брокер сообщений – это приложение промежуточного слоя, работающее как шлюз прикладного уровня в системе очередей сообщений.

Как и любое приложение брокер сообщений может вызываться с помощью API – прикладного программного интерфейса.

В тоже время к брокер сообщений должен поддерживать какой-нибудь протокол, обеспечивающий сетевое взаимодействие между вызывающей и принимающей сторонами.

В простейшем случае это может быть уникальный протокол, работающий на основе стека TCP/IP.

API с поддержкой Message Oriented Middleware (MOM)

Java Message Service (JMS) API - это API с поддержкой Java Message Oriented Middleware (MOM) для отправки сообщений между двумя или несколькими клиентами. JMS является частью платформы Java, Enterprise Edition.

Подобные цели преследовал проект OpenMAMA (Open Middleware Agnostic Messaging API), который является легковесным уровнем интеграции для систем построенных на основе МОМ.

## **100. Протоколы МОМ. AMQP Advanced Message-Queuing Protocol. Основные компоненты AMQP. Использование системной шины для интеграции приложений, созданных для разных программно-аппаратных платформ.**

Среди таких протоколов можно выделить следующие.

* MQTT (Message Queue Telemetry Transport ) – упрощенный сетевой протокол, работающий поверх протоколов транспортного уровня,
* AMQP (Advanced Message Queuing Protocol) – протокол обмена сообщениями между компонентами системы,
* XMPP- (Extensible Messaging and Presence Protocol) – расширяемый протокол обмена сообщениями и данными о присутствии,
* STOMP (Simple/Streaming Text Oriented Messaging Protocol) – потоковый тексто-ориентированный протокол обмена сообщениями.

AMQP обеспечивает взаимодействие между клиентами и брокерами (промежуточным ПО для обмена сообщениями). Он создан для того, чтобы путем стандартизации сообщений предоставить возможность широкому кругу различных приложений и систем работать вместе независимо от их внутренней структуры.

Брокер – это приложение, реализующее модель AMQP, которое принимает соединения клиентов для маршрутизации сообщений и т.п.

Сообщение – это единица передаваемых данных (включая полезные данные и атрибуты сообщения).

Потребитель – приложение, которое получает сообщения из очередей.

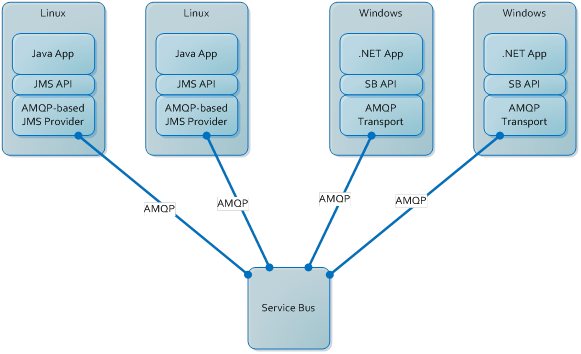
Производитель – приложение, которое отправляет сообщения в очередь через обмен.

Примечание: AMQP не определяет полезную нагрузку сообщений, потому протокол может передавать разные типы данных.

**Использование системной шины для интеграции приложений созданных для разных программно-аппаратных платформ**

Поддержка протокола AMQP 1.0 в Azure Service Bus означает, что с эффективным двоичным протоколом предоставляемые сервисной шиной возможности очередей и обмена сообщениями с публикацией/подпиской у брокера можно использовать на различных платформах.

Пример сценария развертывания, показывающий межплатформенный обмен сообщениями с использованием служебной шины и протокола AMQP 1.0



## **101. Коммуникации, ориентированные на передачу потоков. Понятие среды переноса информации. Поток данных. Простые и комплексные потоки данных. Асинхронный и синхронный режимы передачи потоков данных. Изохронный режим передачи.**

Виды связи RPC, RMI, МОМ – все они основываются на обмене дискретными данными.

Для данных этого типа временные характеристики их доставки не влияют на корректность, а только на производительность обработки.

Коммуникации на основе потоков предполагают, что содержимое сообщений должно быть доставлено с определенной скоростью, так как от этого зависит корректность его представление.

Например, аудио- и видео данные.

Для воспроизведения звука необходимо, чтобы выборки аудиопотока проигрывались в том же порядке, в котором они представлены в потоке данных, и с интервалами ровно по 1/44100 с. Воспроизведение с другой скоростью создаст неверное представление об исходном звуке.

**Понятие среды переноса информации**

Под средой понимается то, что несет информацию. Сюда могут входить среды передачи и хранения, среда представления, например монитор, и т. д. Важнейшая характеристика среды — способ представления

Так, текст обычно кодируется символами ASCII или Unicode. Изображения могут быть представлены в различных форматах, например GIF или JPEG. Аудиопотоки в компьютерных системах могут кодироваться, например, с помощью 16-битных выборок, использующих импульсно-кодовую модуляцию.

В непрерывгюй среде представления временные соотношения между различными элементами данных лежат в основе корректной интерпретации смысла данных.

В отличие от непрерывной среды дискретная среда представления, характеризуется тем, что временные соотношения между элементами данных не играют фундаментальной роли в правильной интерпретации данных. Типичными примерами дискретной среды являются представления текста и статических изображений, а также объектный код и исполняемые файлы.

**Поток данных**

Для обмена критичной ко времени передачи информацией распределенные системы обычно предоставляют поддержку потоков данных. Поток данных есть не что иное, как последовательность элементов данных.

Потоки данных применимы как для дискретной, так и для непрерывной среды представления.

Так, каналы UNIX или соединения TCP/IP представляют собой типичные примеры дискретных потоков данных (байт-ориентированных).

Воспроизведение звукового файла обычно требует непрерывного потока данных между файлом и устройством воспроизведения.

Временные характеристики важны для непрерывных потоков данных. Для поддержания временных характеристик часто приходится выбирать между различными режимами передачи.

**Асинхронный и синхронный режимы передачи**

В асинхронном режиме передачи элементы данных передаются в поток один за другим, но на их дальнейшую передачу никаких ограничений в части временных характеристик не вводится. Это традиционный вариант для дискретных потоков данных. Так, файл можно преобразовать в поток данных, но выяснять точный момент окончания передачи каждого элемента данных чаще всего бессмысленно.

В cuнхронном режиме передачи для каждого элемента в потоке данных определяется максимальная задержка сквозной передачи.

Если элемент данных был передан значительно быстрее максимально допустимой задержки, это не важно. Так, например, датчик может с определенной частотой измерять температуру и пересылать эти измерения по сети оператору.

**Изохронный режим передачи**

При изохронном режиме передачи необходимо, чтобы все элементы данных передавались вовремя. Это означает, что передача данных ограничена максимально, а также минимально допустимыми задержками, также известными под названием предельного дрожания.

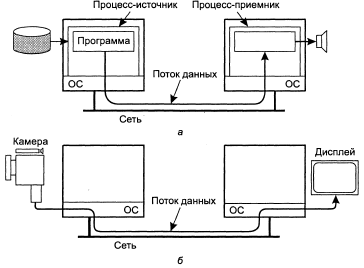
Изохронный режим передачи, в частности, представляет интерес для распределенных систем мультимедиа, поскольку играет значительную роль в воспроизведении аудио- и видеоинформации.

## **102. Передача потока данных по сети. Групповая передача потоков. Качество обслуживания потоков данных. Алгоритм корзины элементарных пакетов.**

При передаче данных через сеть процесс-источник может читать данные из аудиофайла и пересылать их, байт за байтом, по сети. Приемник может быть процессом, по мере поступления выбирающим байты и передающим их на локальное устройство звуковоспроизведения.

В распределенных мультимедийных системах можно реализовать прямое соединение между источником и приемником (а).

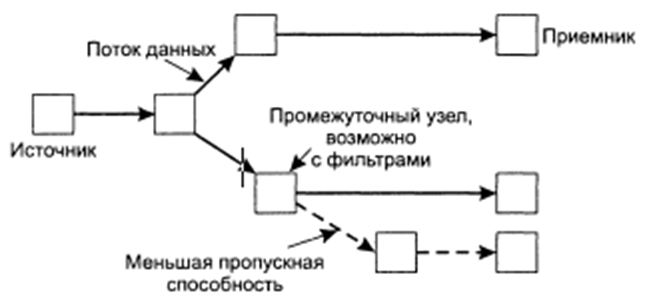
Также , видеопоток, создаваемый камерой, может напрямую передаваться на дисплей (б).



**Групповая передача потоков**

Другая ситуация имеет место в зависимости от того, имеется у нас всего один источник или приемник или мы можем организовать многостороннюю связь.

Наиболее частая ситуация при многосторонней связи — присоединение к потоку данных нескольких приемников. Другими словами, осуществляется групповая рассылка потока данных нескольким получателям.



Главной проблемой групповой рассылки потоков данных являются разные требования разных приемников к качеству потока.

**Потоки данных и качество обслуживания**

Временные зависимости и другие нефункциональные требования обычно выражаются в виде требований к качеству обслуживания (QoS).

Эти требования описывают, что должны сделать базовая распределенная система и сеть для того, чтобы гарантировать, например, сохранение в потоке данных временных соотношений.

Требования QoS для непрерывных потоков данных в основном характеризуются временными диаграммами, объемом и надежностью.

**Алгоритм корзины элементарных пакетов**

Спецификация передачи приведенная выше сформулирована в понятиях алгоритма корзины элементарных пакетов, который описывает, каким образом поток формирует сетевой трафик.

Основная идея состоит в том, что элементарные пакеты генерируются с постоянной скоростью. Элементарный пакет содержит фиксированное число байтов, которые приложение намерено передать по сети. Элементарные пакеты собираются в корзину, емкость которой ограничена.

После переполнения корзины элементарные пакеты просто пропадают. Каждый раз, когда приложение хочет передать в сеть элемент данных размером N, оно должно извлечь из корзины столько элементарных пакетов, чтобы их суммарный размер был не менее N байт. То есть, например, если каждый элементарный пакет N имеет длину k байт, приложение должно удалить из корзины как минимум N/k элементарных пакетов.

# **Тема 15. Синхронизация в распределенных системах**

## **103. Основные механизмы синхронизации в распределенных системах.**

* Синхронизация часов
* Логические часы
* Глобальное состояние
* Алгоритмы голосования
* Взаимное исключение
* Распределенные транзакции

## **104. Синхронизация времени. Три философии (цели) синхронизации часов.**

**Всеми́рное координи́рованное вре́мя (Coordinated Universal Time, UTC)** — стандарт, по которому общество регулирует часы и время. Отличается на целое количество секунд от атомного времени и на дробное количество секунд от всемирного времени UT1. UTC было введено вместо устаревшего среднего времени по Гринвичу (GMT). Новая шкала времени UTC была введена, поскольку шкала GMT является неравномерной шкалой и связана с суточным вращением Земли. Шкала UTC основана на равномерной шкале атомного времени (TAI) и является более удобной для гражданского использования.

**Среднее время по Гринвичу ( Greenwich Mean Time, GMT), или гринвичское время** — среднее солнечное время меридиана, проходящего через прежнее место расположения Гринвичской королевской обсерватории около Лондона.

Ранее, до 1972 года, гринвичское время, GMT, считалось точкой отсчёта времени в других часовых поясах. Ныне в этом качестве используется всемирное координированное время, оно же UTC.

**Универсальное согласованное время (UTC)**

Международное бюро решило эту проблему, используя потерянные секунды (leap seconds) всякий раз, когда разница между временем TAI и солнечным временем возрастает до 800 мс. Эта коррекция позволила перейти к системе, основанной на постоянных секундах TAI, в которой, однако, соблюдается соответствие с периодичностью очевидно видимого движения солнца.

Она называется универсальным согласованным временем {Universal Coordinated Time, UTC). UTC — это основа всей системы хранения времени в наши дни. Оно, по существу, заменило старый стандарт — среднее время по Гринвичу {Greenwich mean time), которое основывалось на астрономических наблюдениях и расчетах.

**Три философии (цели) синхронизации часов**

* Попытаться обеспечить как можно более точную синхронизацию с реальным временем UTC.
* Попытаться обеспечить максимально возможную синхронизацию узлов друг с другом, даже в ущерб синхронизацией с UTC.
* Обеспечить синхронизацию достаточную для обеспечения правильного взаимодействия узлов друг с другом на основе сохранения правильного порядка обмена сообщениями. В этом случае говорят **логических часах** РС.

## **105. Алгоритмы синхронизации часов. Алгоритм Беркли. Алгоритм Кристиана. Задержка при передаче значения времени по сети. Множественные внешние источники точного времени. Отметки времени Лампорта. Алгоритмы синхронизации часов**

**Алгоритмы синхронизации часов**

Network Time Protocol (NTP):

* Цель: обеспечить синхронизацию всех часов по UTC в пределах 1-50мс. Используется в сетях TCP/IP.
* Для синхронизации используется иерархия пассивных серверов NTP

Алгоритм Беркли:

* Цель: обеспечить синхронизацию часов узлов друг с другом (внутренняя синхронизация)
* Для синхронизации используются активные сервера времени периодически опрашивающие узлы.

Reference broadcast synchronization (RBS) (Опорная широковещательная синхронизация)

* Цель: обеспечить синхронизацию часов узлов друг с другом в беспроводной сети

**Алгоритм Беркли**

Демон времени (исполняющийся процесс на сервере времени) запрашивает у всех остальных машин значения их часов (а).

Демон получает ответы машин (б).

Демон времени сообщает всем, как следует подвести их часы (в)

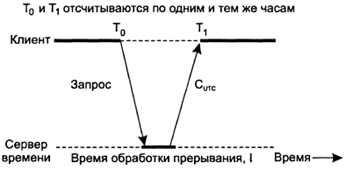
**Алгоритм Кристиана**

Машину с приемником WWV назовем *сервером времени.*

Периодически, гарантировано не реже, чем каждые 2r·Δt с, каждая машина посылает серверу времени сообщение, запрашивая текущее время.

Когда отправитель получает ответ, он может просто выставить свои часы в значение CUTC . Однако такой алгоритм имеет две проблемы:

* Главную – время идет только вперед.
* Второстепенную – ответное сообщение поступает с не нулевой задержкой.



**Задержка при передачи значения времени по сети**

По Кристиану, метод решения проблемы состоит в измерении величины задержки передачи по сети.

Для повышения точности Кристиан предложил производить не одно измерение, а серию. Все измерения, в которых разность *Т1-Т0 превосходит некоторое* пороговое значение, отбрасываются как ставшие жертвами перегруженной сети, а потому недостоверные. Оценка делается по оставшимся замерам, которые могут быть усреднены для получения наилучшего значения.

**Множественные внешние источники точного времени**

Для систем, которым необходима особо точная синхронизация по UTC, можно предложить использование нескольких приемников WWV, GEOS или других источников UTC.

Однако из-за врожденной неточности самих источников времени и флуктуации на пути сигнала лучшее, что могут сделать операционные системы, — это установить интервал, в который попадает UTC. В основном различные источники точного времени будут порождать различные диапазоны, и машины, к которым они присоединены, должны прийти к какому-то общему соглашению.

Чтобы достичь этого соглашения, каждый процессор с источником UTC может периодически делать широковещательную рассылку своих данных, например, точно в начале каждой минуты по UTC.

Но печально то, что задержка между посылкой и приемом будет зависеть от длины кабеля и числа маршрутизаторов, через которые должен будет пройти пакет. Эти значения различны для каждой пары (источник UTC, процессор). Будут также играть свою роль и другие факторы, так что точной синхронизации добиться не удастся.

**Отметки времени Лампорта**

Для синхронизации логических часов Лампорт определил отношение под названием «происходит раньше». Выражение а→b читается как «а происходит раньше b» и означает, что все процессы согласны с тем, что первым происходит событие а, а позже — событие b, Отношение «происходит раньше» непосредственно исполняется в двух случаях.

* Если а и b — события, происходящие в одном и том же процессе, и а происходит раньше, чем b, то отношение а → Ь истинно.
* Если а — это событие отсылки сообщения одним процессом, а b — событие получения того же сообщения другим процессом, то отношение а → Ь также истинно.

## **106. Алгоритмы выборов. Алгоритм голосования. Алгоритм забияки. Кольцевой алгоритм.**

**Алгоритмы голосования**

Многие распределенные алгоритмы требуют, чтобы один из процессов был координатором, инициатором или выполнял другую специальную роль.

Обычно не важно, какой именно процесс выполняет эти специальные действия, главное, чтобы он вообще существовал.

Роль координатора может выполнять любой процесс.

Любой процесс может инициировать процедуру выборов.

Каждый процесс имеет некоторый уникальный номер.

В общем, алгоритмы голосования пытаются найти процесс с максимальным номером и назначить его координатором. Алгоритмы различаются способами поиска координатора.

**Алгоритм забияки**

Когда один из процессов замечает, что координатор больше не отвечает на запросы, он инициирует голосование. Процесс, например Р, проводит голосование следующим образом.

1. Р посылает всем процессам с большими, чем у него, номерами сообщение

ГОЛОСОВАНИЕ.

2. Далее возможно два варианта развития событий:

если никто не отвечает, Р выигрывает голосование и становится координатором;

если один из процессов с большими номерами отвечает, он становится координатором, а работа Р на этом заканчивается.

В любой момент процесс может получить сообщение ГОЛОСОВАНИЕ от одного из своих коллег с меньшим номером. По получении этого сообщения получатель посылает отправителю сообщение ОК, показывая, что он работает и готов стать координатором.

Затем получатель сам организует голосование. В конце концов, все процессы, кроме одного, отпадут, этот последний и будет новым координатором. Он уведомит о своей победе посылкой всем процессам сообщения, гласящего, что он новый координатор и приступает к работе.

**Кольцевой алгоритм**

Этот алгоритм голосования основан на использовании логического кольца - процессы физически или логически упорядочены, так что каждый из процессов знает, кто его преемник.

Когда один из процессов обнаруживает, что координатор не функционирует, он строит сообщение *ГОЛОСОВАНИЕ, содержащее* его номер процесса, и посылает его своему преемнику.

Если преемник не работает, отправитель пропускает его и переходит к следующему элементу кольца или к следующему, пока не найдет работающий процесс.

На каждом шаге отправитель добавляет свой номер процесса к списку в сообщении, активно продвигая себя в качестве кандидата в координаторы. В конце концов, сообщение вернется к процессу, который начал голосование.

В этот момент тип сообщения изменяется на *КООРДИНАТОР и вновь* отправляется по кругу, на этот раз с целью сообщить всем процессам, кто стал координатором (элемент списка с максимальным номером) и какие процессы входят в новое кольцо.

## **107. Понятие критической области при работе с ресурсами. Распределенные транзакции. Взаимные исключения в распределенных системах. Понятие критической области при работе с ресурсами**

Системы, состоящие из множества процессов, обычно проще всего программировать, используя критические области.

Когда процесс нуждается в том, чтобы считать или обновить совместно используемые структуры данных, он сначала входит в критическую область, чтобы путем взаимного исключения убедиться, что ни один из процессов не использует одновременно с ним общие структуры данных.

В однопроцессорных системах критические области защищаются семафорами, мониторами и другими конструкциями подобного рода.

**Распределенные транзакции**

Концепция транзакций тесно связана с концепцией взаимных исключений.

Алгоритмы взаимного исключения обеспечивают одновременный доступ не более чем одного процесса к совместно используемым ресурсам.

Транзакции, в общем, также защищают общие ресурсы от одновременного доступа нескольких параллельных процессов.

Однако транзакции могут и многое другое:

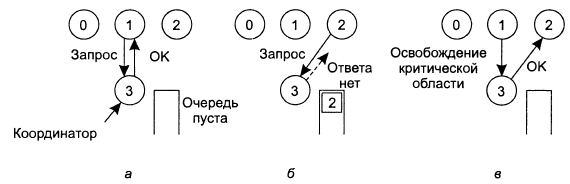
* Они превращают процессы доступа и модификации множества элементов данных в одну атомарную операцию;
* Если процесс во время транзакции решает остановиться на полпути и повернуть назад, все данные восстанавливаются с теми значениями и в том состоянии, в котором они были до начала транзакции.

**Взаимные исключения в распределенных системах**

Наиболее простой способ организации взаимных исключений в распределенных системах состоит в том, чтобы использовать методы их реализации, принятые в однопроцессорных системах.

Один из процессов выбирается координатором (например, процесс, запущенный на машине с самым большим сетевым адресом).

Каждый раз, когда этот процесс собирается войти в критическую область, он посылает координатору сообщение с запросом, в котором уведомляет, в какую критическую область он собирается войти, и запрашивает разрешение на это. (а).



Другой процесс, 2, запрашивает разрешение на вход в ту же самую область (*б). Координатор знает, что в этой критической* области уже находится другой процесс, и не дает разрешения на вход.

Когда процесс 1 выходит из критической области, он сообщает об этом координатору, который разрешает доступ процессу 2 (в).

## **108. Алгоритмы взаимных исключений. Распределенный алгоритм исключения. Алгоритм маркерного кольца.**

**Распределенный алгоритм**

Рассматриваемый алгоритм требуют наличия полной упорядоченности событий в системе. То есть в любой паре событий, например отправки сообщений, должно быть однозначно известно, какое из них произошло первым. Алгоритм Лампорта, является одним из способов введения подобной упорядоченности и может быть использован для расстановки отметок времени распределенных взаимных исключений.

Когда процесс собирается войти в критическую область, он создает сообщение, содержащее имя критической области, свой номер и текущее время. Затем он отсылает это сообщение всем процессам, концептуально включая самого себя. Посылка сообщения, как предполагается, надежная, то есть на каждое письмо приходит подтверждение в получении. Вместо отдельных сообщений может быть использована доступная надежная групповая связь.

Когда процесс получает сообщение с запросом от другого процесса, действие, которое оно производит, зависит от его связи с той критической областью, имя которой указано в сообщении.

Можно выделить три варианта:

1. Если получатель не находится в критической области и не собирается туда входить, он отсылает отправителю сообщение *ОК,*
2. *Если получатель находится в критической области, он не отвечает, а помещает* запрос в очередь.
3. Если получатель собирается войти в критическую область, но еще не сделал этого, он сравнивает метку времени пришедшего сообщения с меткой времени сообщения, которое он отослал. Выигрывает минимальное. Если пришедшее сообщение имеет меньший номер, получатель отвечает посылкой сообщения *ОК. Если его собственное сообщение имеет меньшую отметку* времени, получатель ставит приходящие сообщения в очередь, ничего не посылая при этом.

После посылки сообщения-запроса на доступ в критическую область процесс приостанавливается и ожидает, что кто-нибудь даст ему разрешение на доступ. После того как все разрешения получены, он может войти в критическую область.

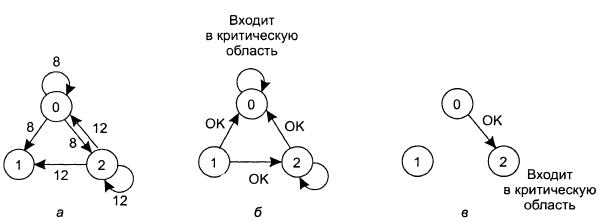
Когда он покидает критическую область, то отсылает сообщения *ОК всем* процессам в их очереди и удаляет все сообщения подобного рода из своей очереди.

**Работа алгоритма распределенного исключения**

Представим себе, что два процесса пытаются одновременно войти в одну и ту же критическую область (а).

Процесс 0 имеет меньшую отметку времени и потому выигрывает (б).

Когда процесс 0 завершает работу с критической областью, он отправляет сообщение ОК, и теперь процесс 2 может войти в критическую область *(в)*

**

**Алгоритм маркерного кольца**

Программно создается логическое кольцо, в котором каждому процессу назначается его положение в кольце. При инициализации кольца процесс О получает маркер, или токен (token).

Маркер циркулирует по кольцу. Он передается от процесса k процессу k+ 1 (это модуль размера кольца) сквозными сообщениями. Когда процесс получает маркер от своего соседа, он проверяет, не нужно ли ему войти в критическую область.

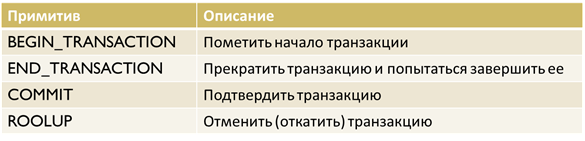
Если это так, он входит в критическую область, выполняет там всю необходимую работу и покидает область. После выхода он передает маркер дальше. Входить в другую критическую область, используя тот же самый маркер, запрещено.

Если процесс, получив от соседа маркер, не заинтересован во входе в критическую область, он просто передает этот маркер дальше.

## **109. Распределенные транзакции. Модель транзакций. Классификация транзакций. Способы реализации транзакций. Закрытое рабочее пространство. Журнал с упреждающей записью.**

Свойство транзакций «все или ничего» — это лишь одно из характерных свойств транзакции. Говоря более конкретно, транзакции:

* атомарны (atomic) — для окружающего мира транзакция неделима;
* непротиворечивы (consistent) — транзакция не нарушает инвариантов системы;
* изолированы (isolated) — одновременно происходящие транзакции не влияют друг на друга;
* долговечны (durable) — после завершения транзакции внесенные ею изменения становятся постоянными.

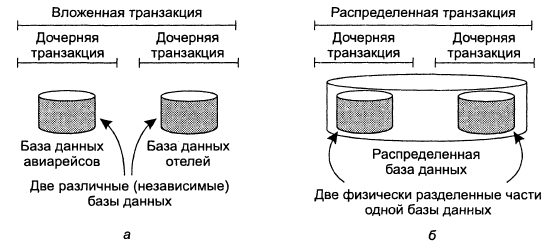


**Классификация транзакций**

Плоская транзакция - серия операций, удовлетворяющая свойствам ACID.

Плоские транзакции имеют одно ограничение - они не могут давать частичного результата в случае завершения или прерывания. Другими словами, сила атомарности плоских транзакций является в то же время и их слабостью.

* Вложенные транзакции: Транзакция верхнего уровня может разделяться на дочерние транзакции, работающие параллельно, на различных машинах, для повышения производительности или упрощения программирования.
* Распределенные транзакции: так как вложенные транзакции (плоские) работают с данными, распределенными по нескольким машинам, то такие транзакции известны под названием *распределенных транзакций (distributed transactions).*

**

**Способы реализации транзакций**

Обычно используются два метода:

* **Закрытое рабочее пространство**.
* **Журнал с упреждающей записью.**

**Закрытое рабочее пространство**

Концептуально, когда процесс начинает транзакцию, он получает закрытое рабочее пространство, содержащее все файлы, к которым он хочет получить доступ.

ока транзакция не завершится или не прервется, все операции чтения и записи будут происходить не в файловой системе, а в **закрытом рабочем пространстве**.

Это утверждение прямо приводит нас к первому методу реализации — созданию для процесса, в момент начала транзакции, закрытого рабочего пространства.

**Журнал с упреждающей записью**

Согласно этому методу файлы действительно модифицируются там же, где находятся, но перед тем, как какой-либо блок действительно будет изменен, в журнал заносится запись со сведениями о том, какая транзакция вносит изменения, какой файл и блок изменяются, каковы прежние и новые значения.

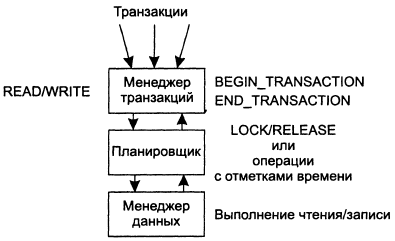
Только после успешной записи в журнал изменения вносятся в файл.

## **110. Управление параллельным выполнением транзакций. Менеджеры транзакций. Изолированность транзакций. Алгоритм двухфазной блокировки. Пессимистическое упорядочение по отметкам времени. Управление параллельным выполнением транзакций**

Цель управления параллельным выполнением транзакций состоит в том, чтобы позволить нескольким транзакциям выполняться одновременно, но таким образом, чтобы набор обрабатываемых элементов данных (например, файлов или записей базы данных) оставался непротиворечивым.

Непротиворечивость достигается в результате того, что доступ транзакций к элементам данных организуется в определенном порядке так, чтобы конечный результат был таким же, как и при выполнении всех транзакций последовательно.

Управление параллельным выполнением лучше всего можно понять в терминах трех менеджеров, организованных по уровням:



На самом верхнем уровне находится *менеджер транзакций.*

*На среднем уровне – планировщик.*

*В самом низу – менеджер данных.*

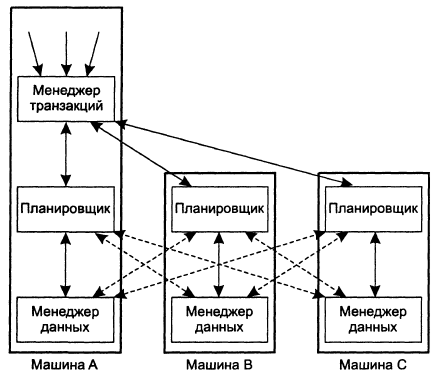
**Менеджеры транзакций**

Менеджер транзакций отвечает, прежде всего, за атомарность и долговечность. Он обрабатывает примитивы транзакций, преобразуя их в запросы к планировщику.

Каждая машина в этом случае имеет своих планировщика и менеджера данных, которые совместно обеспечивают гарантии непротиворечивости локальных данных.

Каждая транзакция обрабатывается одним менеджером транзакций. Последний работает с планировщиками отдельных машин.

В зависимости от алгоритма управления параллельным выполнением транзакций планировщик также может работать с удаленными менеджерами данных.



**Изолированность**

Основная задача алгоритмов управления параллельным выполнением — гарантировать возможность одновременного выполнения многочисленных транзакций до тех пор, пока они изолированы друг от друга. Это значит, что итоговый результат их выполнения будет таким же, как если бы эти транзакции выполнялись одна за другой в определенном порядке.

**Двухфазная блокировка**

Самый старый и наиболее широко используемый алгоритм управления параллельным выполнением транзакций — это *блокировка (locking).*

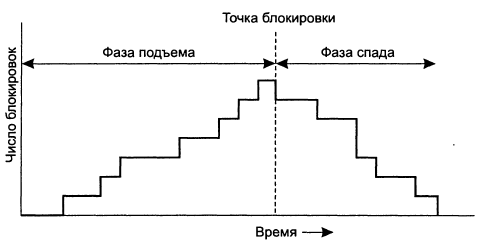
При *двухфазной блокировке (2PL),* планировщик сначала, на *фазе подъема, устанавливает* все необходимые блокировки, а затем, на *фазе спада,* снимает их.

При 2PL выполняются три правила:

- Проверка не конфликтует ли эта операция с другими уже заблокированными операциями.

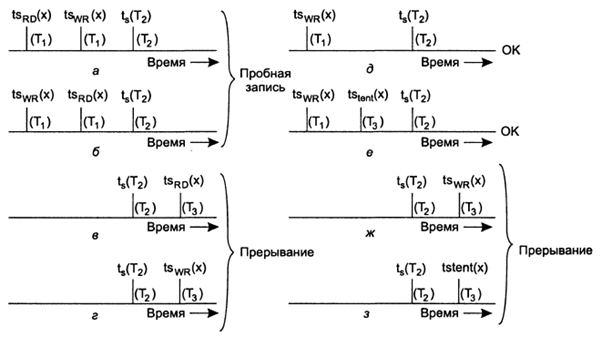
- Планировщик никогда не снимает блокировку с элемента х, если менеджер сообщает, что он выполняет операцию с х.

- Когда планировщик снимает блокировку с операции установленную по требованию транзакции Т, он никогда не делает новую блокировку по требованию этой транзакции.



Доказано, что если все транзакции используют двухфазную блокировку, любой план, сформированный путем перекрытия этих транзакций, сериализуем. В этом причина популярности двухфазной блокировки.

**Пессимистическое упорядочение по отметкам времени**



## **111. Безопасность в РИС: конфиденциальность, целостность, угрозы, аутентификация, авторизация, аудит, доверенная вычислительная база. Правила и механизмы защиты РИС.**

Распределенная информационная система: информационная система, элементы (компоненты, узлы) которой пространственно отделены друг от друга, но пользователю система представляется как единое целое.

**конфиденциальность** - Свойство информации быть недоступной и закрытой для неавторизованного индивидуума, логического объекта или процесса.

**целостность** - означает, что данные полны, условие того, что данные не были изменены при выполнении любой операции над ними, будь то передача, хранение или представление.

**Аутентификация** используется для подтверждения личности зарегистрированного пользователя. **Проверка подлинности** – это процесс проверки учетных данных: идентификатора пользователя (имени, адреса электронной почты, номера телефона) и пароля.

**авторизация** — предоставление этому лицу возможностей в соответствие с положенными ему правами или проверка наличия

**Доверенная вычислительная система** – система, в которой в доверенной среде функционируют защищенные информационные технологии