Moscow Coding School

Python как первый язык

Преподаватель: Захарчук Сергей Сергеевич

Сегодня

- Как работают процессы.
- Как работают потоки в питоне.
- Создание потока.
- Блокировки (Lock).
- Всё о проектировании WEB- приложений
- git

Как работают процессы

Модуль subprocess, который упрощает управление другими программами, передавая им опции командной строки и организуя обмен данными через каналы (pipe)

Как работают процессы

- Рассмотрим в пример два скрипта – parent.py и child.py. Запускается parent.py. Child.py выступает в роли аргумента command, который передается в запускаемый процесс. У этого процесса есть стандартный вход, куда мы передаем два аргумента – поисковое слово и имя файла. Мы запустим два экземпляра программы child.py, каждый экземпляр будет искать слово word в своем файле – это будут файлы исходников самих программ. Запись на стандартный вход осуществляет модуль subprocess. Каждый процесс пишет результат своего поиска в консоль. В главном процессе мы ждем, пока все child не закончат свою работу.

Как работают потоки в питоне

Если нужно, чтобы ваше приложение выполняло несколько задач в одно и то же время, то можете воспользоваться потоками (threads). Потоки позволяют приложениям выполнять в одно и то же время множество задач. Многопоточность (multi-threading) важна во множестве приложений, от примитивных серверов до современных сложных и ресурсоёмких игр.

Как работают потоки в питоне

Если нужно, чтобы ваше приложение выполняло несколько задач в одно и то же время, то можете воспользоваться потоками (threads). Потоки позволяют приложениям выполнять в одно и то же время множество задач.

Многопоточность (multi-threading) важна во множестве приложений, от примитивных серверов до современных сложных и ресурсоёмких игр.

Как работают потоки в питоне

Когда в одной программе работают несколько потоков, возникает проблема разграничения доступа потоков к общим данным. Предположим, что есть два потока, имеющих доступ к общему списку. Первый поток может делать итерацию по этому списку:

```
>>> for x in L
        ...
```

Как работают потоки в питоне

Когда в одной программе работают несколько потоков, возникает проблема разграничения доступа потоков к общим данным. Предположим, что есть два потока, имеющих доступ к общему списку. Первый поток может делать итерацию по этому списку:

```
>>> for x in L
        ...
```

а второй в этот момент начнет удалять значения из этого списка. Тут может произойти все что угодно: программа может упасть, или мы просто получим неверные данные.

Как работают потоки в питоне

Доступ к заблокированному списку будет иметь только один поток, второй будет ждать, пока блокировка не будет снята.

deadlock

Применение блокировки порождает другую проблему – дедлок (deadlock) – мертвая блокировка. Пример дедлока: имеется два потока и два списка. Первый поток блокирует первый список, второй поток блокирует второй список. Первый поток изнутри первой блокировки пытается получить доступ к уже заблокированному второму списку, второй поток пытается проделать то же самое с первым списком. Получается неопределенная ситуация с бесконечным ожиданием.

deadlock

Решение- политика очередности блокировок

Блокировка первого списка идет всегда первой, а уже потом идет блокировка второго списка

deadlock

Ещё проблема, что несколько потоков могут одновременно ждать доступа к уже заблокированному ресурсу и при этом ничего не делать. Каждая программа всегда имеет главный управляющий поток.

deadlock

Интерпретатор питона использует внутренний глобальный блокировщик (GIL), который позволяет выполняться только одному потоку.

Это сводит на нет преимущества многоядерной архитектуры процессоров. Для многопоточных приложений, которые работают в основном на дисковые операции чтения/записи, это не имеет особого значения, а для приложений, которые делят процессорное время между потоками, это является серьезным ограничением.

Создание потока

- Для создания потоков используется стандартный модуль threading. Есть два варианта создания потоков:
- вызов функции

threading.Thread()

- вызов класса

threading.Thread

Создание потока

```
import threading
import time
def clock(interval):
        while True:
                print("The time is %s" % time.ctime())
                time.sleep(interval)
t = threading.Thread(target=clock, args=(15,))
t.daemon = True
t.start()
```

Создание потока

```
import threading
import time
class ClockThread(threading.Thread):
        def __init__(self,interval):
                threading.Thread.__init__(self)
                self.daemon = True
                self.interval = interval
        def run(self):
                while True:
                        print("The time is %s" % time.ctime())
                        time.sleep(self.interval)
t = ClockThread(15)
t.start()
```

Создание потока

Для управления потоками существуют методы:

- start() – дает потоку жизнь.
- run() –этот метод представляет действия, которые должны быть выполнены в потоке.
- join([timeout]) – поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван. Параметр timeout (число с плавающей точкой) позволяет указать время ожидания (в секундах), по истечении которого приостановленный поток продолжает свою работу независимо от завершения потока, чей метод join был вызван. Вызывать join() некоторого потока можно много раз. Поток не может вызвать метод join() самого себя. Также нельзя ожидать завершения еще не запущенного потока.

Создание потока

- getName() – возвращает имя потока.
- setName(name) – присваивает потоку имя name.
- isAlive() – возвращает истину, если поток работает (метод run() уже вызван).
- isDaemon() – возвращает истину, если поток имеет признак демона.
- setDaemon(daemonic) – устанавливает признак daemonic того, что поток является демоном.
- activeCount() – возвращает количество активных в настоящий момент экземпляров класса Thread. Фактически это len(threading.enumerate()).
- currentThread() – возвращает текущий объект-поток, т.е. соответствующий потоку управления, который вызвал эту функцию.
- enumerate() – возвращает список активных потоков.

Блокировки (Lock)

Создадим три потока, каждый из которых будет считывать стартовую страницу по указанному Web-адресу.

Иимеется глобальный ресурс – список урлов – url_list – доступ к которому будет блокироваться с помощью блокировки threading.Lock(). Объект Lock имеет методы:

- acquire([blocking=True]) – делает запрос на запирание замка. Если параметр blocking не указан или является истиной, то поток будет ожидать освобождения замка.

Если параметр не был задан, метод не возвратит значения.

Если blocking был задан и истинен, метод возвратит True (после успешного овладения замком).

Если блокировка не требуется (т.е. задан blocking=False), метод вернет True, если замок не был заперт и им успешно овладел данный поток. В противном случае будет возвращено False.

- release() – запрос на отпирание замка.
- locked() – возвращает текущее состояние замка (True – заперт, False – открыт).

Блокировки (Lock)

см. threads.py

Блокировки (Lock)

```
import threading from urllib
import urlopen
class WorkerThread(threading.Thread):
        def __init__(self,url_list,url_list_lock):
                super(WorkerThread,self).__init__()
                self.url_list=url_list self.url_list_lock=url_list_lock def run(self):
                while (1):
                        nexturl = self.grab_next_url()
                        if nexturl==None:break
                        self.retrieve_url(nexturl)
```

Блокировки (Lock)

```
def grab_next_url(self):
                    self.url_list_lock.acquire(1)
                    if len(self.url_list)<1: nexturl=None
                    else: nexturl = self.url_list[0]
                    del self.url_list[0]
                    self.url_list_lock.release()
                    return nexturl
```

Блокировки (Lock)

```
def retrieve_url(self,nexturl):
        text = urlopen(nexturl).read()
        print(text )
        print('######################## %s ############################' %
nexturl)
```

Блокировки (Lock)

```
url_list=['http://linux.org.ru','http://kernel.org','http://python.org']
url_list_lock = threading.Lock()

workerthreadlist=[]

for x in range(0,3):
        newthread = WorkerThread(url_list,url_list_lock)
        workerthreadlist.append(newthread)

        newthread.start()

for x in range(0,3):
        workerthreadlist[x].join()
```

Архитектура клиент-сервер

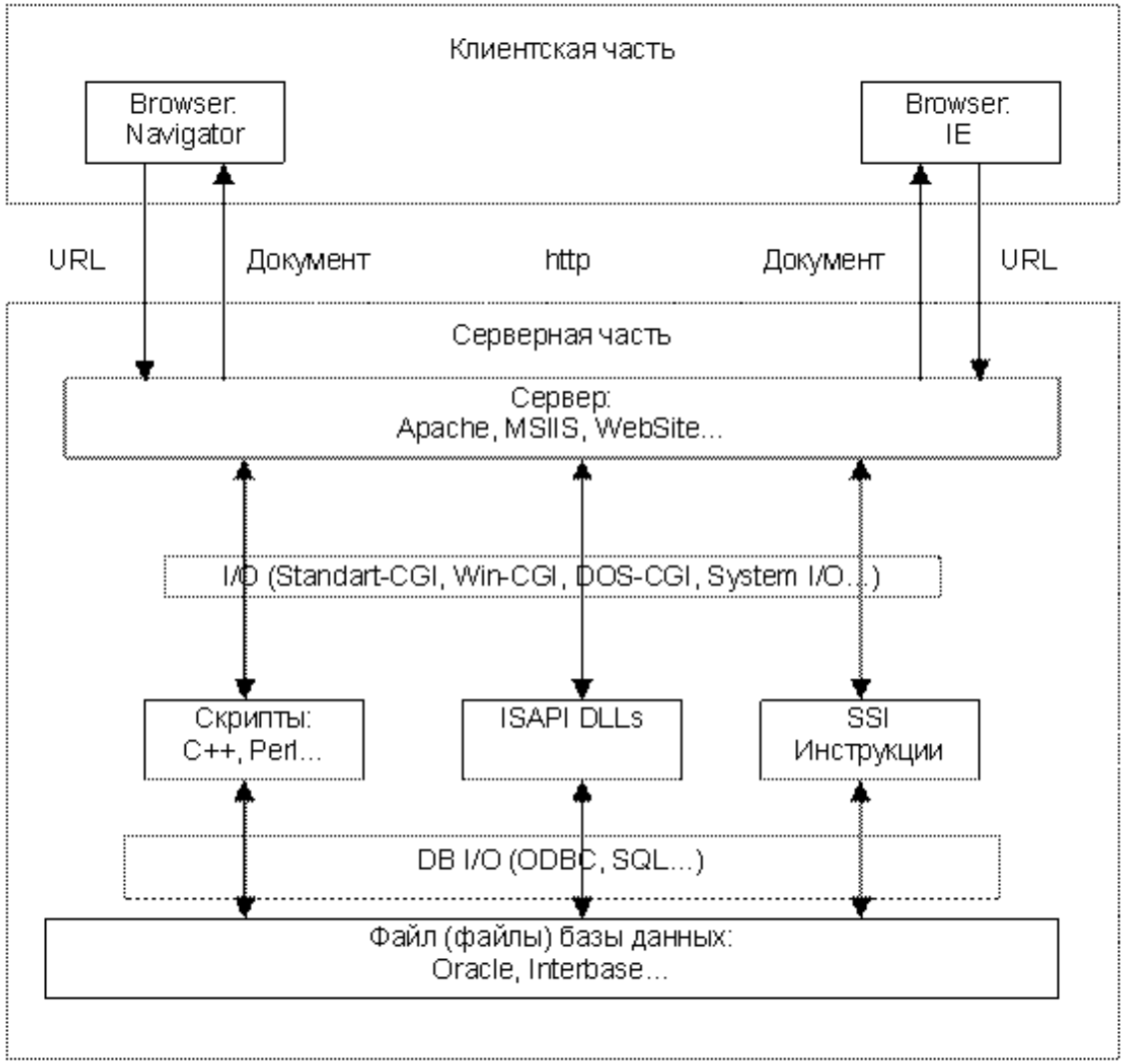
Сеть Интернет организована по схеме клиент-сервер. В классическом случае данная схема функционирует следующим образом:

- клиент формирует и посылает запрос на сервер баз данных;
- сервер производит необходимые манипуляции с данными, формирует результат и передаёт его клиенту;
- клиент получает результат, отображает его на устройстве вывода и ждет дальнейших действий пользоватля.

Цикл повторяется, пока пользователь не закончит работу с сервером.

В сервисе WWW для передачи информации применяется протокол HTTP (HyperText Transmition Protocol).

Схема клиент-сервер WWW-HTTP



Транзакции в HTTP

Основные транзакции в HTTP:

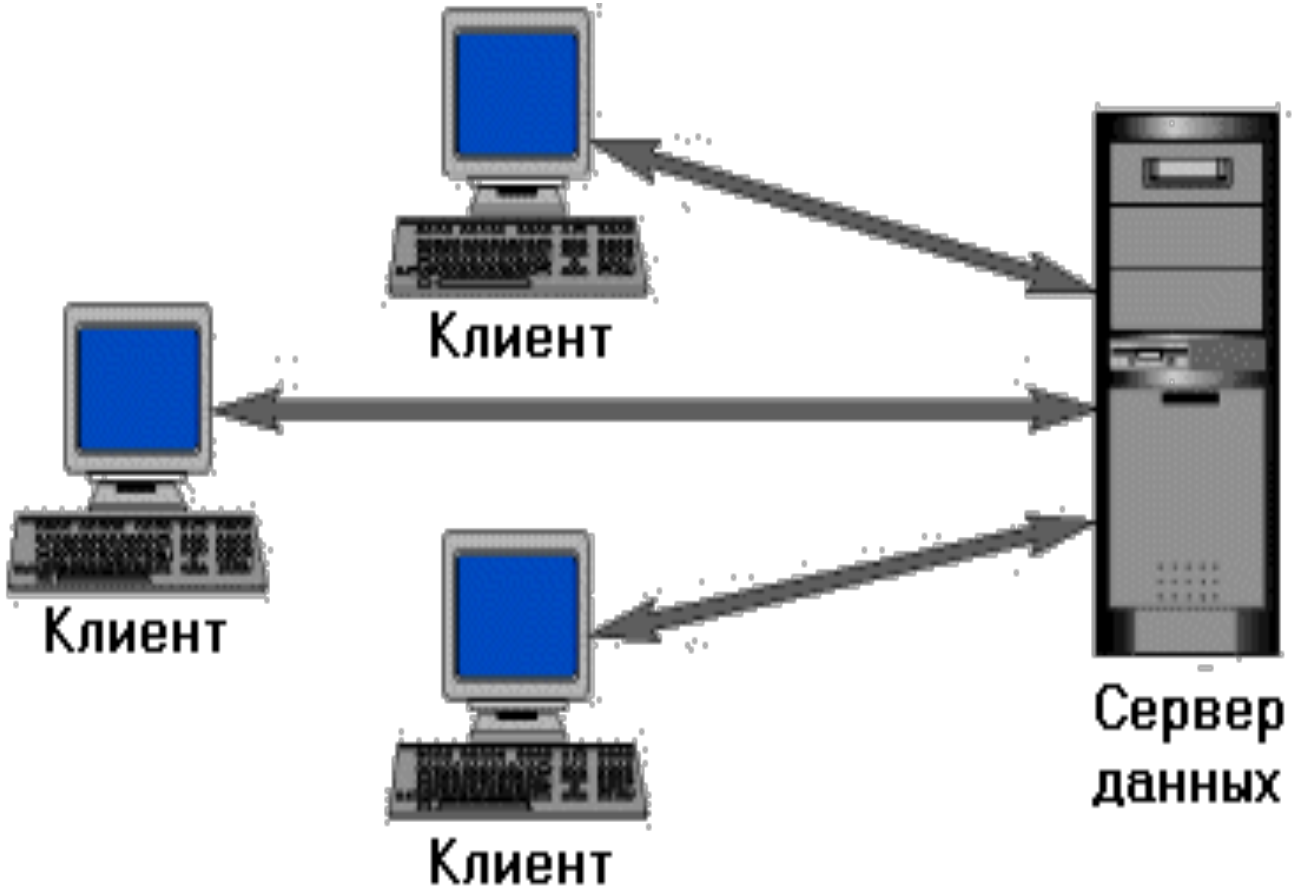
1. Браузер декодирует первую часть URL (Universal Resource Locator) и устанавливает соединение с сервером.
2. Браузер передает остальную часть URL на сервер.
3. Сервер определяет по URL путь и имя файла.
4. Сервер пересылает указанный файл браузеру.
5. Сервер прерывает соединение.
6. Браузер отображает документ.

При данных транзакциях сервер не имеет никакой информации о состоянии браузера, т.е. HTTP можно считать "однонаправленным" протоколом, и взаимодействовать с сервером возможно только через механизм URL, это создает трудности при реализации клиентской части.

Распределение функций в архитектуре "клиент-сервер"

- Основная задача клиентского приложения – это обеспечение интерфейса с пользователем, т. е. ввод данных и представление результатов в удобном для пользователя виде, и управление сценариями работы приложения.
- Основные функции серверной СУБД – обеспечение надежности, согласованности и защищенности данных, управление запросами клиентов, быстрая обработка SQL-запросов.
- В двухзвенной архитектуре вся логика работы приложения (прикладные задачи, бизнес-правила) распределяется между двумя процессами: клиентом и сервером.

Двухзвенная архитектура "клиент-сервер"



Двухзвенная архитектура "клиент-сервер"

1. Архитектура **"толстый клиент – тонкий сервер"**: большая часть функций приложения решалась клиентом, сервер занимался только обработкой SQL-запросов.

Архитектура "толстый" клиент имеет следующие недостатки:

- сложность администрирования;
- усложняется обновление ПО, поскольку его замену нужно производить одновременно по всей системе;
- усложняется распределение полномочий, так как разграничение доступа происходит не по действиям, а по таблицам;
- перегружается сеть вследствие передачи по ней необработанных данных;
- слабая защита данных, поскольку сложно правильно распределить полномочия.

Двухзвенная архитектура "клиент-сервер"

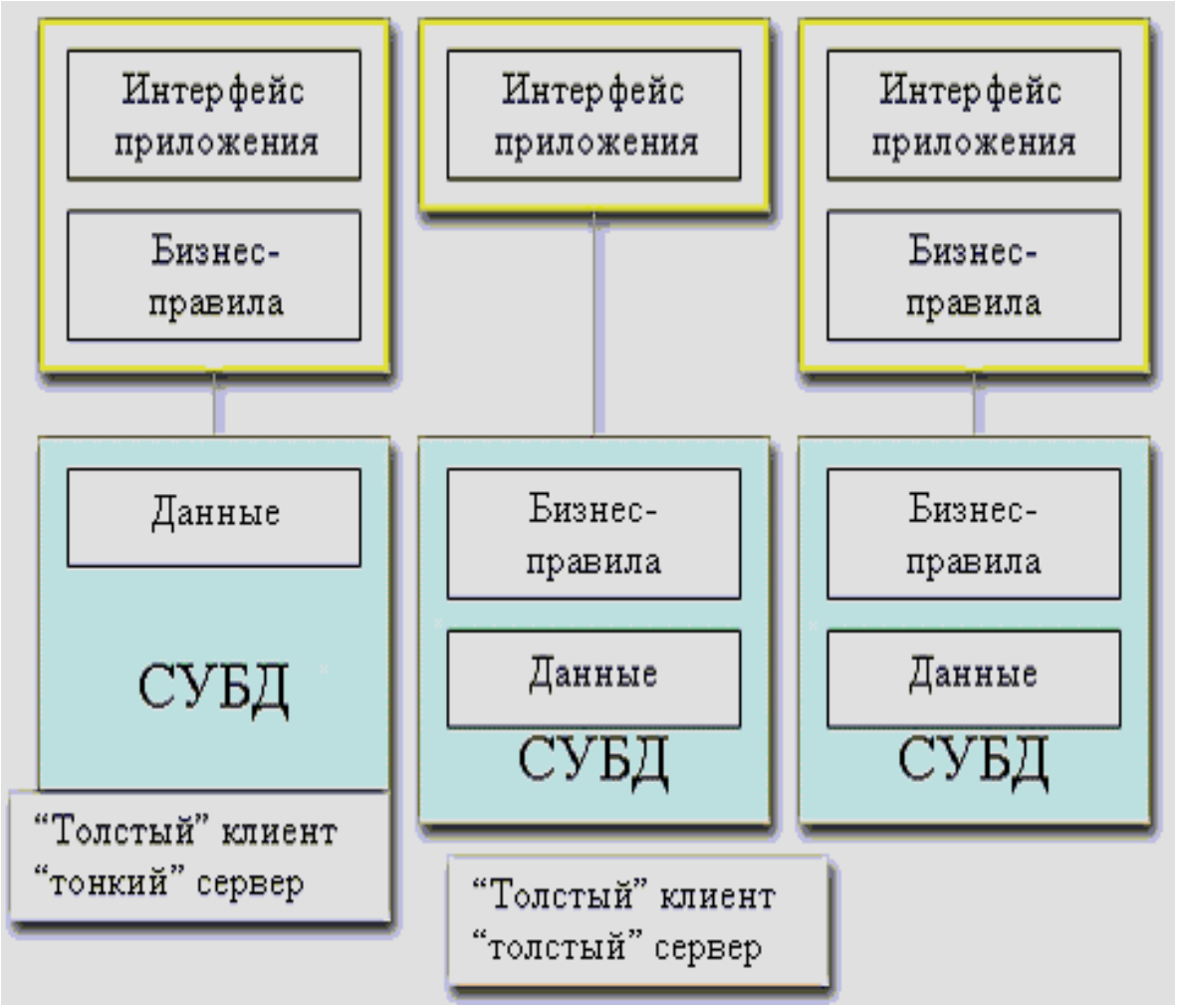
2. Архитектура **"тонкий клиент – толстый сервер"**: использование на сервере хранимых процедур (stored procedure - откомпилированные программы с внутренней логикой работы), привело к тенденции переносить все большую часть функций на сервер. Хранимые процедуры реализовывали часть бизнес-логики и гарантировали выполнение операции в рамках единой транзакции. Такое решение имеет очевидные преимущества, например его легче поддерживать, т. к. все изменения нужно вносить только в одном месте – на сервере.

Архитектура "толстый" сервер имеет следующие недостатки:

- программы, написанные на СУБД-языках, обычно работают недостаточно надежно; ошибка в них может привести к выходу из строя всего сервера баз данных;
- получившиеся таким образом программы полностью непереносимы на другие системы и платформы.

Для решения перечисленных проблем используются многоуровневые (три и более уровней) архитектуры клиент-сервер.

Распределение функций в архитектуре "клиент-сервер"



Многозвенная архитектура "клиент-сервер"

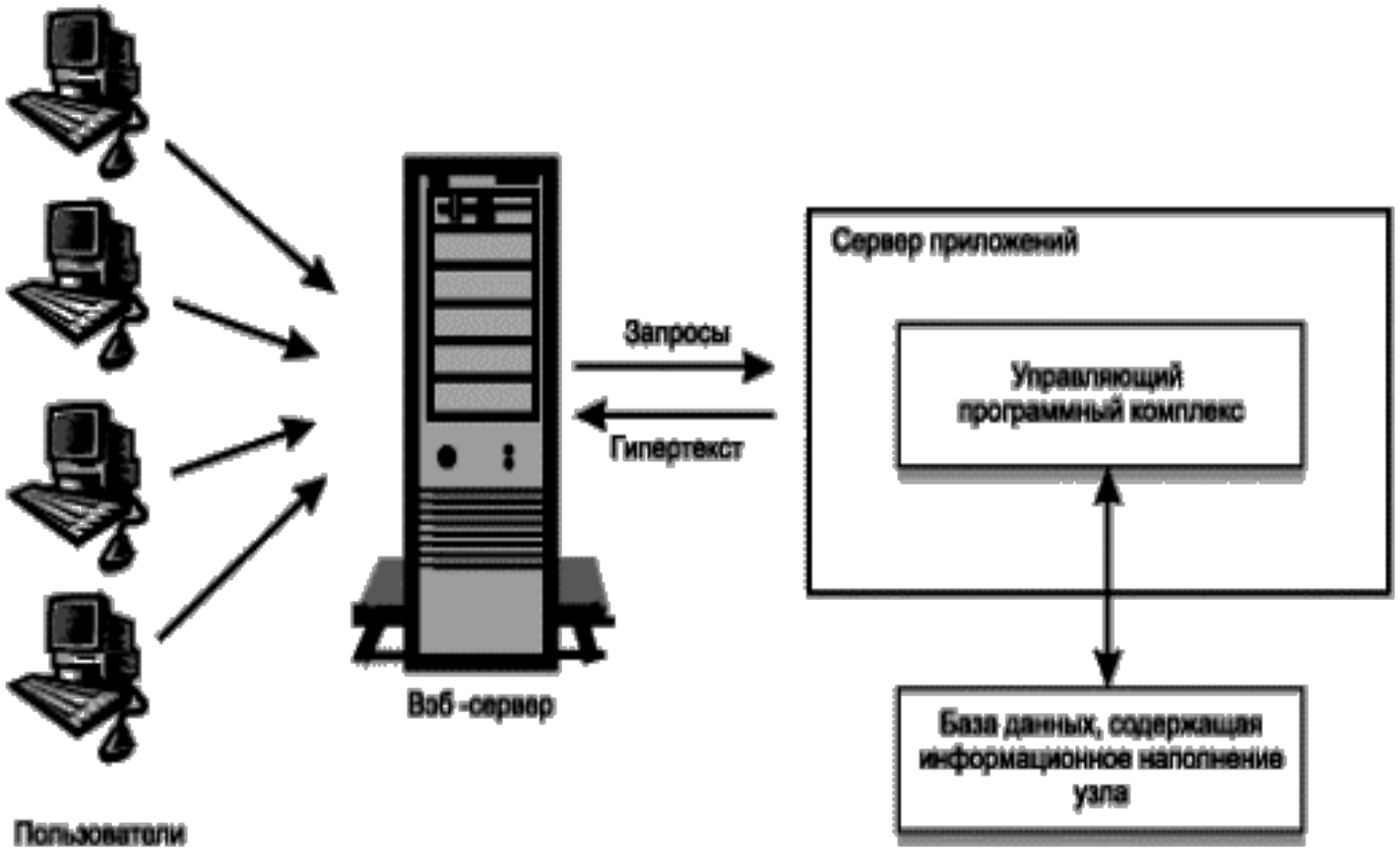
- **Трехзвенная и многозвенная архитектуры "клиент-сервер"**: выполнение прикладных задач и бизнес-правил осуществляется отдельным компонентом приложения (или нескольким компонентам), которые могут работать на специально выделенном компьютере – сервере приложений.

- Сервер приложений обрабатывает следующие компоненты:

1. презентационная логика (Presentation Layer - PL) – предназначена для работы с данными пользователя;
2. бизнес-логика (Business Layer - BL) – предназначена для проверки правильности данных, поддержки ссылочной целостности;
3. логика доступа к ресурсам (Access Layer - AL) – предназначена для хранения данных.

- Подход Remote Data Access (RDA) подразумевает объединение в клиентском приложении PL и BL (однако в случае необходимости выполнения каких-либо изменений в клиентском приложении придется менять исходный код), а серверная часть представляет собой сервер баз данных, реализующий AL.

Трехзвенная архитектура "клиент-сервер"



Многозвенная архитектура "клиент-сервер"

- Любая информационная система, построенная на основе клиент-серверных технологий, должна содержать следующие компоненты:

1. шлюз-сервер, управляющий правами доступа к информационной системе;
2. WWW-сервер;
3. сервер баз данных;
4. сервер приложений и(или) сервер обработки транзакций.

- Взаимодействие WWW сервера с базами данных может быть организовано двумя способами:

1. через сервер (менеджер) транзакций;
2. через API интерфейс WWW сервера или сервера приложений.

Менеджер транзакций

Менеджеры транзакций позволяют одному серверу приложений одновременно обмениваться данными с несколькими серверами баз данных.

Хотя серверы Oracle имеют механизм выполнения распределенных транзакций, но если пользователь хранит часть информации в БД Oracle, часть в БД Informix, а часть в текстовых файлах, то без менеджера транзакций не обойтись.

МТ используется для управления распределенными разнородными операциями и согласования действий различных компонентов информационной системы.

Первые менеджеры транзакций появились в начале 70-х гг. (например, CICS); с тех пор они незначительно изменились идеологически, но весьма существенно - технологически.

Наибольшие идеологические изменения произошли в коммуникационном менеджере, так как в этой области появились новые объектно-ориентированные технологии (CORBA, DCOM и т.д.).

Менеджер транзакций

Менеджер транзакций – это программа или комплекс программ, с помощью которых можно согласовать работу различных компонентов информационной системы.

Логически МТ делится на несколько частей:

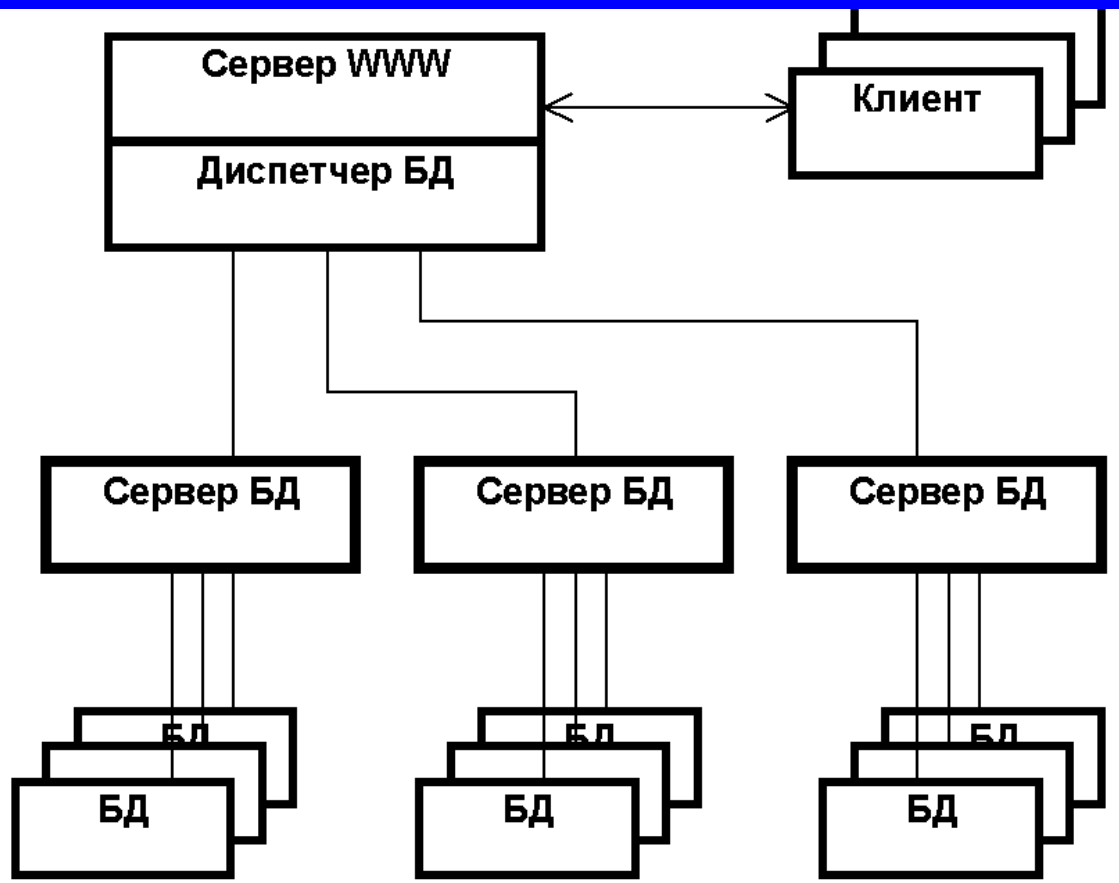
- коммуникационный менеджер (Communication Manager) – контролирует обмен сообщениями между компонентами информационной системы;
- менеджер авторизации (Authorisation Manager) – обеспечивает аутентификацию пользователей и проверку их прав доступа;
- менеджер транзакций (Transaction Manager) – управляет распределенными операциями;
- менеджер ведения журнальных записей (Log Manager) – следит за восстановлением и откатом распределенных операций;
- менеджер блокировок (Lock Manager) – обеспечивает правильный доступ к совместно используемым данным.

Обычно коммуникационный менеджер объединен с авторизационным, а менеджер транзакций работает совместно с менеджерами блокировок и системных записей. Причем такой менеджер редко входит в комплект поставки, поскольку его функции (ведение записей, распределение ресурсов и контроль операций), как правило, выполняет сама база данных (например, Oracle).

Многозвенная архитектура "клиент-сервер"

- Распределенная информационная система представляется в виде трех-четырехуровневой структуры с разграничением функций на каждом уровне и фиксацией протоколов межуровневого потока данных.
- Разграничение на логически замкнутые функциональные уровни необходимо для возможности их реализации на разных физических серверах и добавления в дальнейшем новых возможностей.
- Обмен информацией с уровнем 1 происходит через файловую систему (локальную или сетевую), с уровнем 3 - по протоколам TCP через фиксированный программный порт. В последнем случае для лучшей межплатформенной совместимости данные передаются только в текстовом виде.

Многозвенная архитектура "клиент-сервер"



Многозвенная архитектура "клиент-сервер"

Уровень 1. *Собственно данные* представляют собой обычные файлы данных в формате, необходимом для работы сервера БД. Данные хранятся в виде набора файлов в отдельном каталоге для каждой БД. Кроме собственно данных, каталог может включать информацию о предопределенных форматах для отображения данных и файл заголовка для расширенного названия БД.

Уровень 2. *Сервер баз данных* реализует основные функции выборки информации из БД. Для публичной информационной системы эти функции сводятся к следующим:

- получение запроса с уровня 3;
- логический разбор строки запроса;
- исполнение запроса;
- возврат данных на уровень 3.

Многозвенная архитектура "клиент-сервер"

В соответствии с этим сервер БД обрабатывает следующие запросы.

Информационный – запрос на информацию о конкретной базе данных. Во входном потоке - идентификатор базы данных сервера БД, в выходном - заголовок, количество записей и комментарий указанной БД, описание поле БД.

Словарный – запрос на список ключевых слов с параметрами. Во входном потоке - идентификатор БД, шаблон ключевого слова, порядковый номер ключевого слова, количество слов в выходном буфере, в выходном - список затребованных ключевых слов и их частота.

Форматный – запрос на предоставление списка предопределенных форматов вывода данных. Во входном потоке - идентификатор БД, в выходном - пронумерованный список предопределенных форматов для данной БД.

Основной – запрос на предоставление данных в требуемом формате с параметрами. Во входном потоке - идентификатор БД, строка запроса, номер записи начала вывода, количество записей для вывода, идентификатор формата, в выходном - форматированная выборка из БД.

Служебный – запрос на номер версии сервера БД. В выходном потоке - номер версии текущего сервера БД, пронумерованный список доступных БД, идентификатор внутренней кодировки сервера БД.

Многозвенная архитектура "клиент-сервер"

Уровень 3. Сервер WWW с модулем управления серверами БД - *диспетчер БД* - предназначен для обработки запросов пользователей, формирования запросов к серверам БД и возврата клиентам полученной информации по протоколу HTTP и спецификациям HTML. Оптимальным вариантом является Windows NT + IIS с поддержкой JAVA и ASP (Active Server Pages) ввиду тесной интеграции IIS с операционной системой и возможностью организации многопоточной обработки данных сравнительно простыми и дешевыми средствами. Управляющий модуль (диспетчер БД) может быть реализован в виде динамической библиотеки и (или) набора объектов ASP.

Диспетчер БД выполняет следующие функции:

- хранение и предоставление пользователям текущей информации о доступных БД;
- формирование запросов к серверам БД и возвращение клиентам полученной информации в требуемой кодировке;
- хранение информации о правах доступа на каждую доступную БД и проверка их для каждого пользователя;
- учет и сбор статистики обращений к БД в соответствии с текущими установками;
- синхронизация версий серверов БД и их обновление;
- при наличии уровня 4 передача служебной информации о себе и о поддерживаемых базах данных на уровень 4.

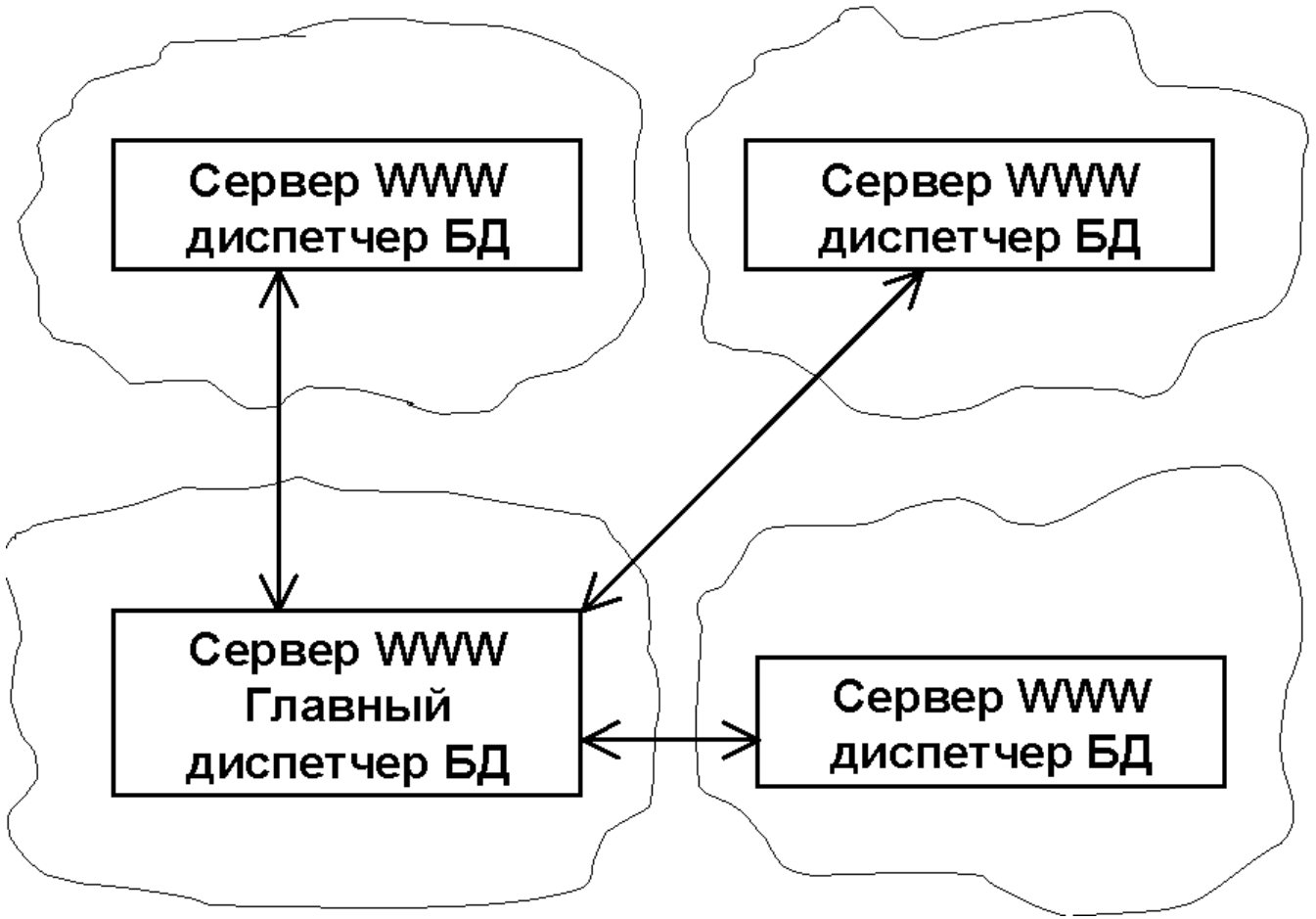
Многозвенная архитектура "клиент-сервер"

Для организации полнофункциональной системы достаточно перечисленных трех уровней. Однако при построении территориально распределенной системы с ярко выраженными районами и ненадежными линиями связи между ними желательно локализовать все три уровня в каждом районе с интеграцией последних на уровне 4.

Уровень 4. *Главный диспетчер (ГД)* информационной системы представляет собой сервер WWW, функционально идентичный серверу уровня 3, но наделенный дополнительной функцией хранения информации о всей информационной системе в целом. В идеальном случае каждый из серверов уровня 3 должен быть готов взять на себя роль главного диспетчера. Основная задача ГД – получить информацию о конфигурации каждого сервера уровня 3 и растиражировать ее по всем серверам.

Таким образом, общая схема распределенной информационной системы состоит из четырех логических уровней.

Интеграция диспетчеров БД на 4 уровне



Основные задачи клиентских и серверных сценариев

- Клиентский сценарий выполняется на компьютере пользователя в процессе взаимодействии с Web-страницей и позволяет решать следующие задачи:

1. верифицировать значения элементов управления формы;
2. реализовать событийные процедуры для элементов управления.

- Серверный сценарий выполняется на Web-сервере до передачи страницы пользователю и позволяет:

1. обеспечить доступ к базе данных и возврат данных пользователю;
2. хранить информацию о состоянии пользователя или сеанса.

Клиентские сценарии

- Клиентский сценарий выполняется на компьютере-клиенте. Программы просмотра снабжены встроенным интерпретатором, который может считывать и выполнять сценарии.
- Основная цель добавления клиентского сценария к Web-странице — создание событийных процедур для элементов управления. Например, событийная процедура будет запускать определенную функцию, когда пользователь нажмет соответствующую кнопку.
- Клиентские сценарии в HTML-странице не компилируются и не шифруются. Поэтому, если посмотреть исходный HTML-код Web-страницы, можно увидеть текст встроенного сценария.
- Чтобы сценарий клиентской части функционировал, программа просмотра должна поддерживать язык, на котором он написан. В противном случае пользователь не получит полного доступа к сценарным средствам Web-страницы.

Серверные сценарии

- Серверный сценарий выполняется в рамках активной страницы на Web-сервере до того, как тот вернет пользователю готовую HTML-страницу. Когда пользователь запрашивает активную серверную страницу, сервер выполняет сценарии и создает HTML-код, который и передается пользователю. В результате пользователь не видит серверного сценария на полученной Web-странице.
- Поскольку серверный сценарий выполняется на Web-сервере, ему доступны все ресурсы сервера – например, базы данных и исполняемые файлы.
- Для работы серверных сценариев Web-сервер должен поддерживать технологию активных страниц; к программе просмотра же не предъявляется никаких дополнительных требований, поскольку Web-клиент в данном случае получает стандартную HTML-страницу. Таким образом, сценарии серверной части не зависят от клиентов.

Реализация клиентских сценариев

Чтобы расширить функциональные возможности Web-страницы средствами клиентских сценариев, исходный текст сценария надо встроить в HTML-страницу в виде ASCII-текста. Встретив ее в тексте страницы, программа просмотра вызывает интерпретатор сценария, который анализирует и выполняет код. Программа просмотра должна поддерживать выполнение сценариев и их интеграцию с элементами управления ActiveX или Java-апплетами, встроенными в HTML-страницу.

Реализация клиентских сценариев

Языки разработки сценариев:

1. **Visual Basic Scripting Edition (VBScript)** – не зависит от регистра символов и совместимо снизу вверх с Visual Basic for Applications. Microsoft Internet Explorer поддерживает VBScript средствами VBScript Interpreter — быстрого кросс-платформенного интерпретатора; лицензию на него бесплатно выдает компания Microsoft.
2. **JavaScript (JScript)** – реализован Microsoft и подобен C: в его основе лежит Java – язык программирования, разработанный компаниями Sun Microsystems и Netscape. JavaScript поддерживают как Netscape Navigator, так и Internet Explorer.

VBScript и JavaScript похожи – как в одном, так и в другом можно определять переменные, создавать процедуры и обращаться к свойствам и методам объектов.

Разница между ними – небольшие отличия в синтаксисе. Ни один из них не компилируется, и оба работают на всех аппаратных платформах. Это интерпретируемые языки, поэтому скорость исполнения определяется возможностями программы просмотра, а не характеристиками самого языка.

Механизмы, реализующие серверную часть обработки данных

1. **Internet Server Application Programming Interface (ISAPI)** – интерфейс программирования приложений сервера Интернета реализуется через механизм библиотек DLL.

Приложения ISAPI являются динамически подключаемыми библиотеками. Такая библиотека с интерфейсными функциями загружается WEB-сервером один раз и остается в памяти, после чего она будет готова отвечать на любое количество запросов. Каждый клиентский запрос обслуживается в отдельном потоке.

Библиотеки DLL работают как часть процесса WEB-сервера, выполняясь в том же пространстве адресов памяти, в котором работает и сам WEB-сервер. Вместо передачи информации в обе стороны в виде файлов, теперь расширения WEB-серверов передают информацию в пределах одного и того же адресного пространства, без необходимости записи в файл. Благодаря этому WEB-приложения стали работать быстрее, с большей эффективностью и с меньшим потреблением ресурсов.

С помощью ISAPI Internet connector возможно взаимодействие с базами данных через драйверы ODBC, также возможна реализация других расширенных функций (создание различных фильтров запросов). Основным средством разработки приложений является Microsoft Visual C++ (также VB, Delphi), который поддерживается Microsoft Internet Information Server.

Механизмы, реализующие серверную часть обработки данных

2. **Server Sides Includes (SSI/SSI+)** – технология динамического формирования документов.

Скрипт (серверные инструкции) находится в HTML файле обычно имеющем расширение sht или shtm, при этом серверные инструкции размещаются между специальными разделителями (tokens), а сами инструкции записаны на языке Cscript. При пересылке такой файл сканируется сервером на наличие SSI инструкций и результат динамически подставляется в посылаемый документ.

SSI реализуется через специальные компоненты (DLL), которые входят в состав сервера. Данная технология опирается на использование разнообразных объектов и компонент (COM, ActiveX и т.п.), работа с которыми ведётся средствами языков VBScript или JavaScript.

Механизмы, реализующие серверную часть обработки данных

3. **Common Gateway Interface (CGI) –** интерфейс общего шлюза реализуется через дополнительные программы (скрипты) на любом из языков программирования высокого уровня (C++, Perl, VisualBasic, Pascal, Java).

По сути CGI – способ взаимодействия Web-программ с браузером пользователя Основа – спецификация набора переменных. С помощью CGI приложений возможно взаимодействие с любыми базами данных через формирование SQL запросов, или другие механизмы; также возможна реализация счетчиков посещений, гостевых книг и других расширений.

CGI обеспечивает способ, посредством которого Web-браузер осуществляет запуск Web-приложения на стороне сервера, результатом работы которого является HTML-страница, посылаемая клиенту. Всякий раз, когда клиент инициирует выполнение CGI-приложения, Web-сервер выполняет отдельную его копию (instance).

Недостатки CGI-приложений

- Для каждого запроса клиента запускается копия Web-приложения на сервере, что резко сокращает производительность сервера при больших и средних нагрузках
- Каждый запрос должен запускать на сервере свой собственный процесс, выделенная ему на сервере область памяти не пересекается с областью памяти приложения web-сервера. И поэтому несколько запросов могут существенно замедлить работу даже умеренно загруженного сервера - ведь ему приходится выполнять такие относительно медленные задачи, как создание файла, запуск отдельного процесса, его выполнение, запись и возвращение другого файла.

Большинство CGI-программ пишется на языке Perl (Practical Extraction and Report Language), который является одним из наиболее гибких языковых средств, служащих для программирования интерфейсов CGI. Изначально Perl предназначался для обработки больших объемов данных и генерации отчетов по обработке этих данных, но за последние несколько лет Perl превратился в полнофункциональный язык программирования.

Недостатки CGI-приложений

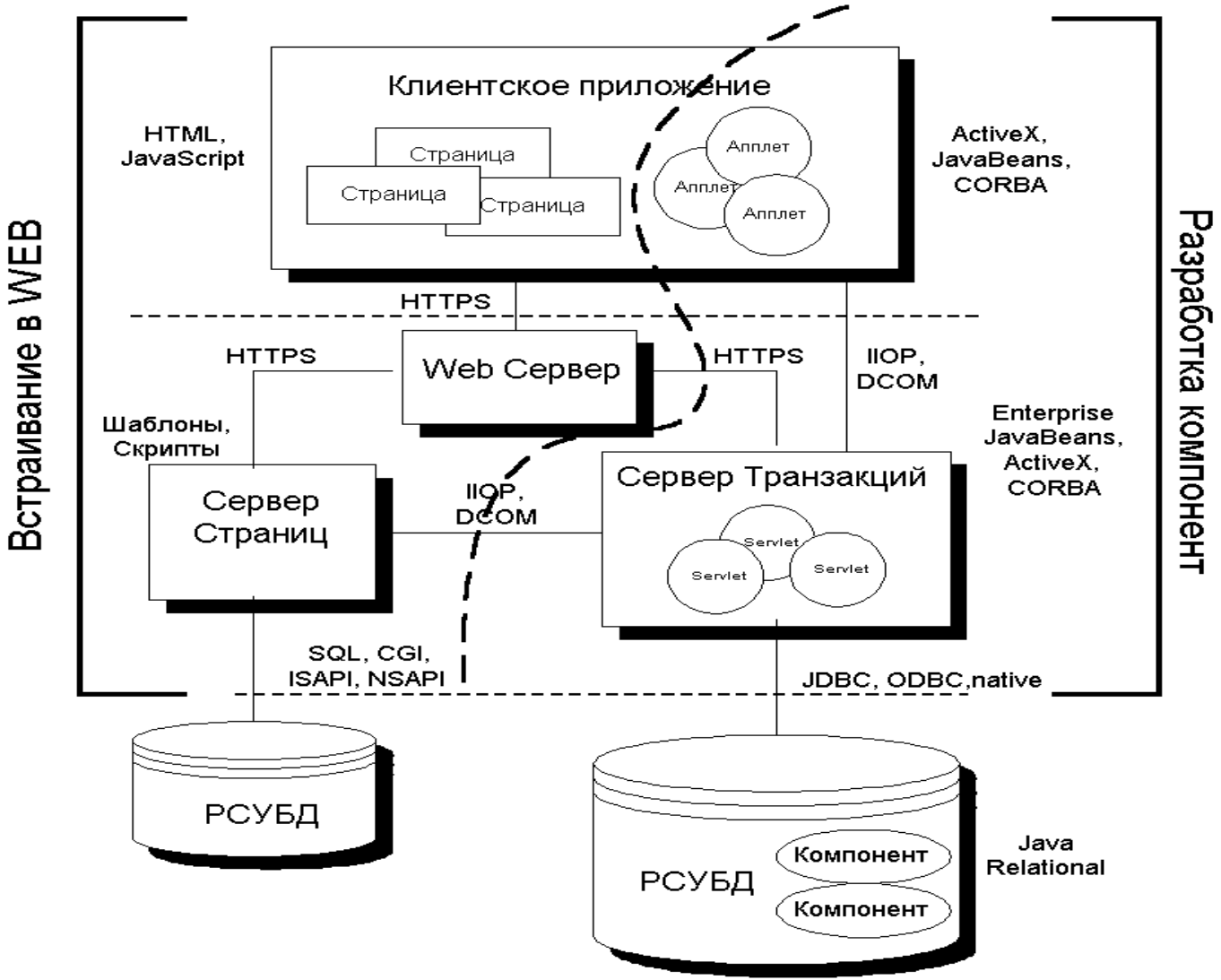
Технология Java – позволяет строить универсальные системы со смешанной архитектурой:

1. апплетами (applets) – приложения, выполняемые на стороне клиента,
2. сервлеты (servlets) – приложения, выполняемые на стороне сервера.

Аплеты пишутся на Java и посылаются по Web как HTML-файлы браузеру, где выполняются как HTML-документы. Существенным преимуществом Java является независимость программ от платформ, на которых программы выполняются. Хотя Java не обязательно выполняется в окне браузера, возможно создание независимых (stand-alone) Java-приложений, которые могут выполняться на компьютере независимо от Интернета.

Фактически программа на языке Java транслируется компилятором в специальный код, называемый байтовым (bytecode), а затем выполняется уже с помощью интерпретатора языка Java. Такое «разделение обязанностей» и позволяет обеспечивать полную независимость Java-кода от конечной платформы, на которой он будет выполняться. Для каждой конкретной платформы имеется свой интерпретатор языка, называемый виртуальной машиной Java (Java Virtual Machine).

Архитектура распределенного приложения



Что такое Git?

Git - программное обеспечение для управления версиями, разработанное Линусом Торвальдсом для использования в управлении разработкой ядра Linux®.

Git – распределенная система контроля версий (DVCS)

Ключевые особенности Git

- Ветвление делается быстро и легко.
- Поддерживается автономная работа; локальные фиксации изменений могут быть отправлены позже.
- Фиксации изменений атомарны и распространяются на весь проект.
- Каждое рабочее дерево в Git содержит хранилище с полной историей проекта.
- Ни одно хранилище Git не является по своей природе более важным, чем любое другое.
- Скорость работы

Репозиторий Git

Git хранит информацию в структуре данных называемой – репозиторий (**repository**).

Репозиторий хранит:

- Набор **commit objects**
- Набор ссылок на **commit objects** называемых **heads.**

Репозиторий хранится в той же директории, что и сам проект в поддиректории *.git*.

Основные отличия от систем с центральным репозиторием (например CVS, SVN):

- Существует только одна директория *.git* в корневой директории проекта
- Репозиторий хранится в файлах рядом с проектом
- Не существует центрального репозитория

Репозиторий Git

Commit objects

Commit objects содержат:

- Набор файлов, отображающий состояние проекта в текущую точку времени
- Ссылки на родительские **commit objects**
- SHA1 имя – 40 символьная строка которая уникально идентифицирует **commit object**. Имя представляющее собой хэш является значимым аспектом *commit* (идентичные commits всегда будут иметь одинаковое имя)

Первый commit в проекте не имеет родительского объекта.

Идея контроля версий состоит в манипулировании графом **commit objects**.

Репозиторий Git

Heads

Head – ссылка на **commit object**. Каждый **head** имеет имя.

В каждом репозитории существует head называемый – *master*.

Репозиторий может содержать любое количество heads.

Выбранный head называют – “***current head***” он имеет синоним – “***HEAD***”

Работа с репозиторием

Создание репозитория выполняется командой: `git init`. После выполнения команды в папке проекта появиться директория .git.

Для выполнения commit необходимо выполнить следующее:

Сказать Git какие файлы необходимо добавить в commit данное действие выполняется командой `git add`. Если файлы не изменились с предыдущего commit то Git добавит их в commit автоматически.

Вызвать команду `git commit` которая создаст commit object.

Команда `git commit -a` добавит все изменившиеся файлы, но не новые файлы.

Работа с репозиторием

Полезные команды:

`git log` – показывает лог commits начиная с *HEAD*

`git status` – показывает какие файлы изменились между текущей стадией и *HEAD*. Файлы разделяются на 3 категории: новые файлы, измененные файлы, добавленные новые файлы

`git mv` – используется для перемещения или переименования файла

`git rm` – удаляет файл из репозитория не затрагивая рабочую копию

`gitk` – визуальная утилита для работы с репозиторием

Получение ссылок на commit выполняется следующим образом:

по SHA1 имя выполнив git log

по 1м символам SHA1 имени

используя head

Работа с branch

Создание branch выполняется следующей командой:

```
git branch branch_name <base_reference>
```

Переключение веток осуществляется командой:

```
git checkout head_name
```

Данная команда выполняет 2 функции:

- Указатель на HEAD **commit object** (head_name)
- Перезапись всех файлов в директории на соответствующие родительскому HEAD (^HEAD) и формирование нового commit.

Полезные сопутствующие команды:

- `git branch` – показывает список HEAD объектов
- `git diff [head1]..[head2]` – показывает изменения между 2мя HEAD
- `git log [head1]..[head2]` – показывает историю изменений

Коллективная работа

Копирование удаленного репозитория осуществляется командой `git clone repository_url`.

Данная команда выполняет следующее:

Создает директорию проекта и инициализирует репозиторий

Копирует все *commit objects* и head ссылки в новый репозиторий

Добавляет удаленные head называемые *origin/[head_name]* соответствующие head в удаленном репозитории

Для работы с удаленной веткой локально необходимо выполнить следующую команду: `git branch [local_branch] [remote-branch]`

Получение изменений из удаленного репозитория выполняется командой: `git fetch [remote-repository-reference]`, по умолчанию это ссылка на *origin*. Данную команду в большинстве случаев заменяет `git pull`.

Коллективная работа

Добавление изменений в удаленный репозиторий выполняется командой `git push [remote-repository-reference] [remote-head-name]`.

После вызова команды происходит следующее:

В удаленный репозиторий добавляются новые *commit object*.

Устанавливается head в удаленном репозитории на тот же commit.

Если выполняется `git push` без аргументов то отправляются все ветки за которыми было установлено слежение.

Ссылки на ресурсы

Бесплатный хостинг репозиториев для open source проектов:

github.com

Официальный сайт

Git: http://git-scm.com/