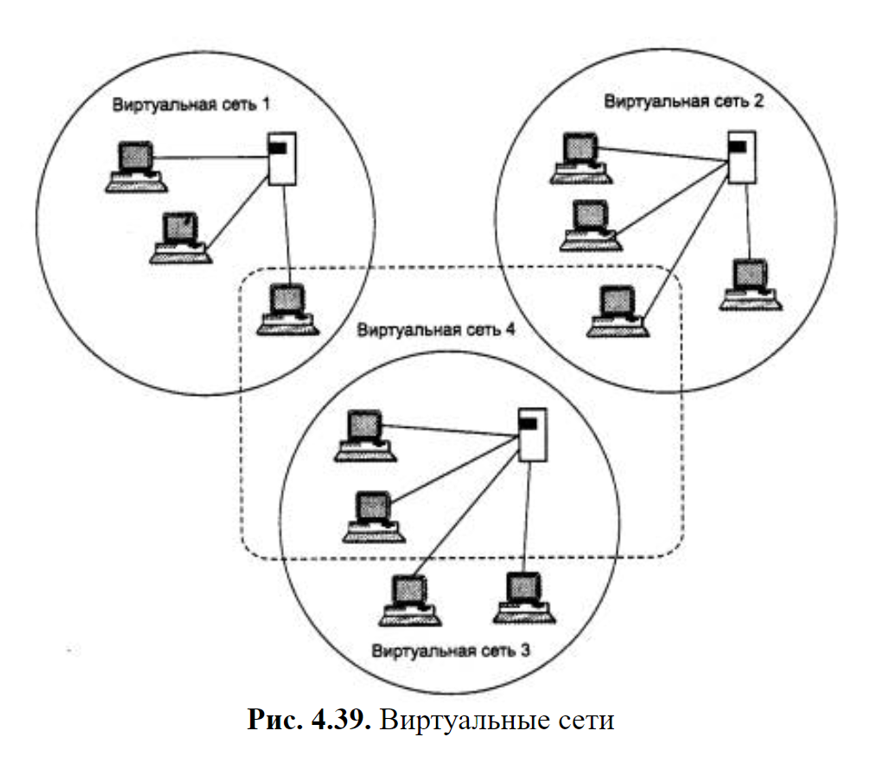
### Понятие виртуальной сети. Виды виртуальных сетевых компонентов.

**Виртуальной сетью** называется группа узлов сети, трафик которой, в том числе и широковещательный, на канальном уровне полностью изолирован от других узлов сети (рис. 4.39).

Это означает, что передача пакетов между разными виртуальными сетями на основании адреса канального уровня невозможна, независимо от типа адреса - уникального, группового или широковещательного. В то же время внутри виртуальной сети пакеты передаются по технологии коммутации, то есть только на тот порт, который связан с адресом назначения пакета.

**Виртуальные сети** *могут* пересекаться, если один или несколько компьютеров входят в состав более чем одной виртуальной сети.



**Назначение технологии виртуальных сетей** состоит в облегчении процесса создания изолированных сетей, которые затем должны связываться с помощью маршрутизаторов, реализующих какой-либо протокол сетевого уровня, например IP.

**Виртуальные сети** подразделяются на две категории:

· VLAN (Virtual Local Area Network, VLAN)

· VPN (англ. Virtual Private Network «*виртуальная* частная *сеть*»)

**Virtual network adapter** (Виртуальный сетевой адаптер) — программный эмулятор сетевой карты, на который устанавливается гостевая ОС. Каждая ВМ может включать до трех сетевых адаптеров; один адаптер включается в состав ВМ сразу при ее создании, если был разрешен любой из вариантов сетевого подключения. (примеры: VMWare, Virtual box)

**Host virtual adapter** (Виртуальный адаптер хоста) — виртуальный Ethernet-адаптер, устанавливаемый на хостовую ОС при установке VMware Workstation. В хостовой ОС семейства Windows он опознается в качестве сетевого адаптера как VMware Virtual Ethernet Adapter. Это виртуальное устройство служит для взаимодействия ВМ с хост-компьютером и включается в состав ВМ, когда для нее задается тип сетевого подключения **Host Only**.

**Bridge** (Мост) — программно реализованный сетевой мост, который позволяет подключать ВМ к реальной локальной сети (Local Area Network, LAN), используя в качестве «посредника» хост-компьютер. Сетевой мост соединяет виртуальный сетевой адаптер с физическим Ethernet-адаптером хост-компьютера. Мост устанавливается во время установки VMware Workstation и включается в конфигурацию ВМ автоматически.

**Virtual switches** (Виртуальные коммутаторы) — эти устройства, подобно физическим коммутаторам, обеспечивают соединение между собой различных узлов сети. Всего в виртуальной сети, создаваемой VMware, может присутствовать до 9 виртуальных коммутаторов. К одному коммутатору можно подключить неограниченное число устройств, если в качестве хостовой ОС используется Windows, и до 32 устройств в хостовой ОС Linux.

**NAT Device** (Устройство преобразования сетевых адресов) — позволяет подключать ВМ к внешней сети (например, к Интернету), когда ВМ невозможно выделить собственный IP-адрес и приходится использовать IP-адрес, назначенный хост-компьютеру.

**DHCP Server** (DHCP-сервер) — программный компонент, обеспечивающий назначение сетевых IP-адресов виртуальным машинам в сети, в которой не используются подключения через мост.

### Трансляция сетевых адресов. Виды NAT.

**NAT (Network Address Translation)** – трансляция сетевых адресов. Технология преобразования IP-адресов внутренней (по другому частной) сети в IP- адреса внешней сети (Интернет)

**Цель создания** – преодоление нехватки адресов IPv4

**Преимущества NAT:**

· Позволяет преодолеть нехватку адресов IPv4

· Легко развернуть и использовать

· Скрывает структуру сети от внешнего мира

**Недостатки NAT:**

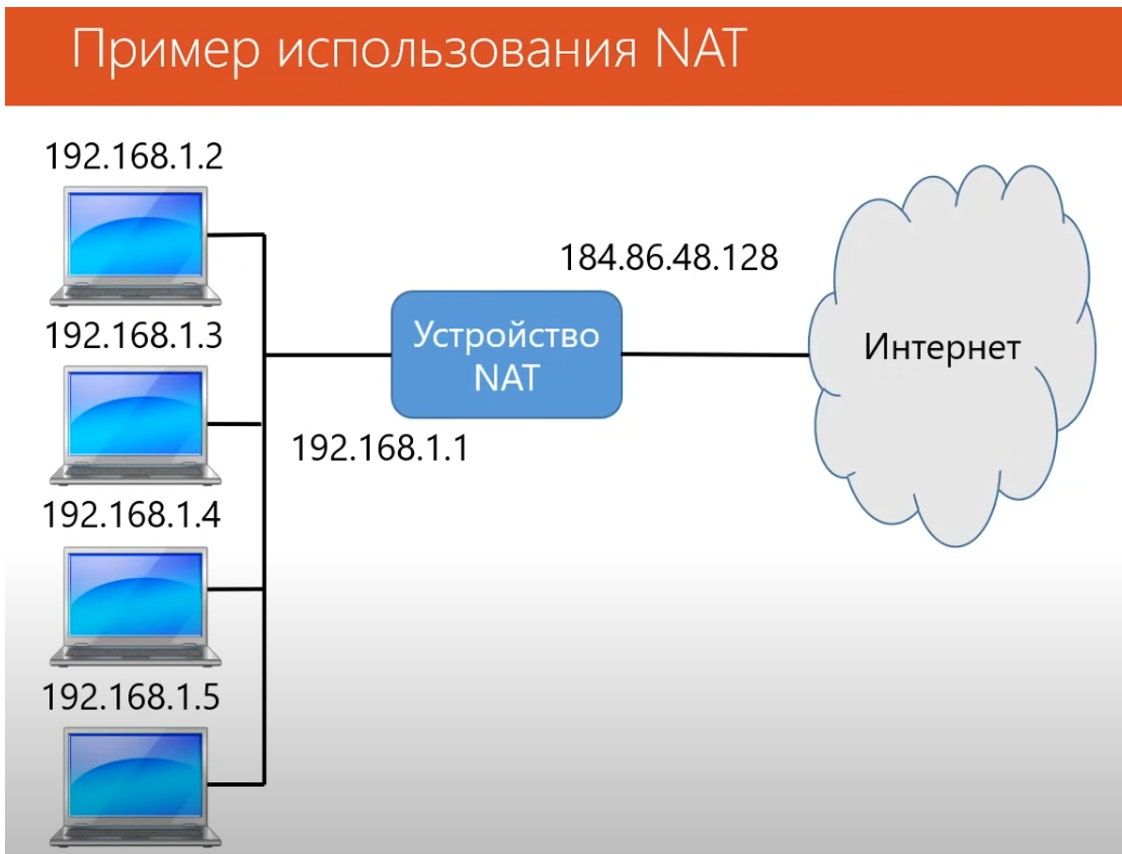
· Нарушение фундаментального принципа построения IP-сетей: каждый компьютер может соединиться с любым другим

· Плохо работают протоколы не устанавливающие соединения

· Некоторые прикладные протоколы работают неправильно FTP

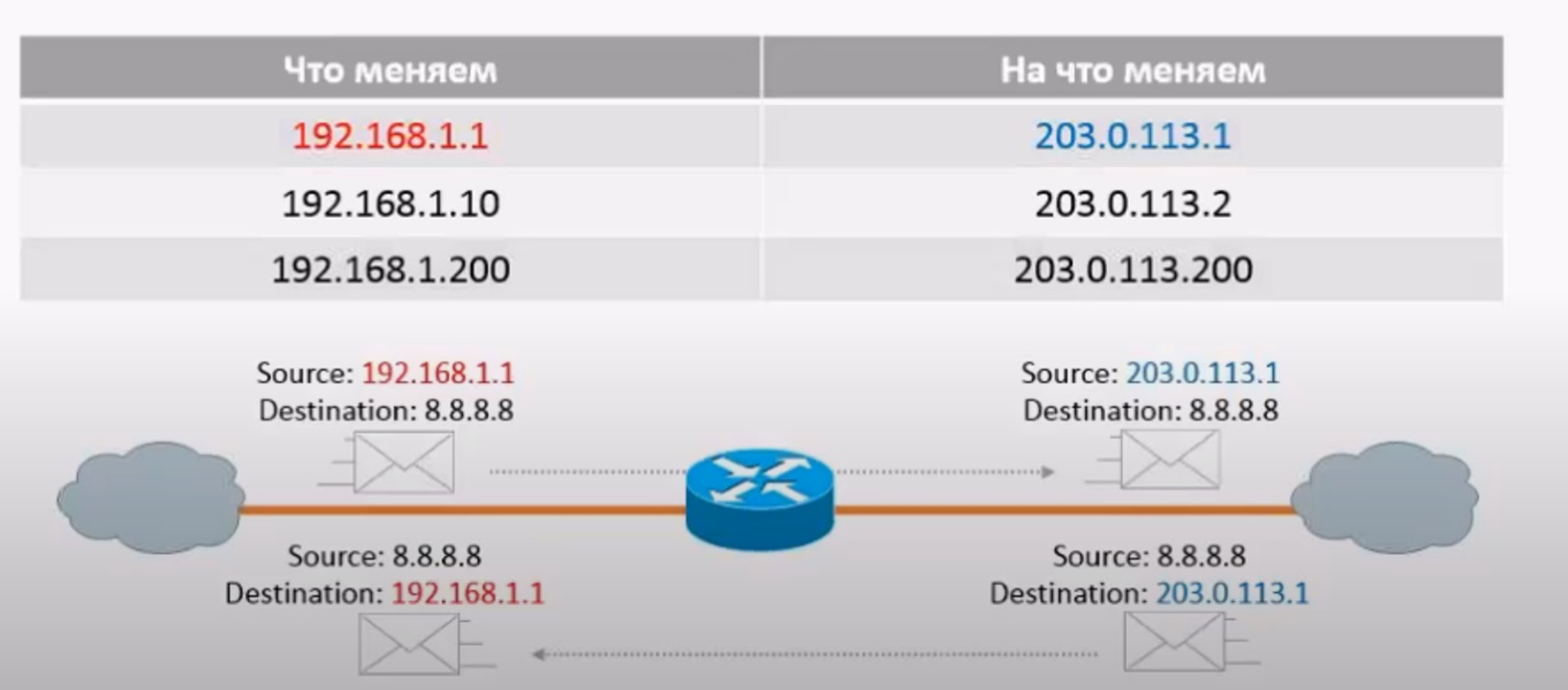
· Нет единого стандарта NAT много разных вариантов

· Нет возможности подключиться к компьютерам во внутренней сети из внешнего мира



У каждого NAT маршрутизатора есть *таблица трансляций*

Каждая трансляция является **двусторонней**

****

## **Типы NAT**

Для понимания, что такое NAT в сети, необходимо разобраться с классификацией трансляции. В частности, по способу сопоставления адресов, бывают такие типы трансляции NAT:

* **Static NAT** — статическая адресная трансляция. Предусматривает сопоставление между глобальными и локальными адресами «один к одному».
* **Dynamic NAT** — динамическая адресная трансляция. Сопоставление адресов осуществляется по принципу «многие ко многим».
* **Port Address Translation (NAT Overload)** — трансляция с использованием портов. Предусматривается многоадресное сопоставление.

### **Статический NAT**

Этот тип NAT использует принцип «один к одному» при сопоставлении локальных и глобальных адресов. Настройки сопоставлений, установленные сетевым администратором, остаются неизменными. При отправке сетевыми устройствами трафика в интернет выполняется преобразование их внутренних локальных адресов в настроенные администратором глобальные внутренние адреса. Для внешних сетей устройства, которые работают во внутренней сети со статическим NAT, имеют общедоступные адреса IPv4.

### 

### **Динамический NAT**

Тип динамического NAT работает с пулом публичных адресов, которые назначаются на основе принципа «первым пришел, первым обслужен». При запросе внутренним устройством доступа к интернету динамическим NAT производится назначение из пула общедоступного адреса IPv4. Как и в случае со статическим, для динамического типа NAT необходимо достаточное число общедоступных адресов, чтобы удовлетворить совокупное число одновременных пользовательских сеансов.

### **Port Address Translation (PAT)**

Это наиболее распространенный тип NAT. При использовании Port Address Translation NAT подключение осуществляет трансляцию нескольких приватных адресов на один или несколько общедоступных. Этот принцип используется в большинстве маршрутизаторов, которые устанавливаются дома у частных абонентов. Адрес назначается провайдером маршрутизатором. При этом одновременный доступ к интернету могут получать несколько домашних пользователей.

### Использование Linux для разработки. Стандартные программные средства.

Для разработчиков прежде всего важно иметь актуальные версии языков программирования, так как есть вероятность того что если программа писалась на более ранней версии языка некоторые функции и модули могут уже некорректно работать или вовсе не существовать.

Нужно также понимать что существует два вида языков программирования:

* компилируемые языки программирования
* интерпретируемые языки программирования

**Компилируемые языки программирования** - язык программирования, исходный код которого преобразуется компилятором в машинный код и записывается в файл с особым заголовком и/или расширением для последующей идентификации этого файла, как исполняемого операционной системой.

Главное **преимущество** таких языков это быстрота обработки исполнения программы

**Примерами компилируемых языков** являются C, C++, Erlang, Haskell и более современные языки, такие как Rust и Go

**Интерпретируемые языки программирования** - язык программирования, в котором исходный код программы не преобразуется в машинный код для непосредственного выполнения центральным процессором (как в компилируемых языках), а исполняется с помощью специальной программы-интерпретатора.

**Преимущества:**

* кроссплатформенность
* рефлексия и интроспекция
* динамическая типизация
* использование динамической области видимости и замыканий
* пошаговое отслеживание выполнения программы
* модификация программы во время исполнения
* меньшие затраты времени на разработку и отладку
* простой способ создания переносимых программ

Примеры таких языков может выступать: Python, PHP, Perl, Ruby

Также для упрощения нахождения ошибок в коде при компиляции и интерпретации исходных текстов программ используется специальный софт называемый **IDE.** IDE нужно также для упрощения работы с синтаксисом программ и вообще разработки.

Самыми известными из IDE можно привести в пример PyCharm, VS Code и др.

В качестве *дополнительного программного обеспечения* для разработчика могут понадобиться bug-tracker’ы, системы контроля версий, всевозможные отладчики и т.д.

### Интерпретатор Python. Использование, версии. Понятие виртуального окружения, настройка, использование.

**Интерпретатор** — это такой модуль, который исполняет другие программы. Когда вы пишете код на языке **Python**, **интерпретатор Python** читает вашу программу и выполняет составляющие ее инструкции.

По сути дела, **интерпретатор** — это слой программной логики между вашим программным кодом и аппаратурой вашего компьютера.

**Виртуальное окружение** — это изолированное окружение среды (в нашем случае **это окружение Python**), которое позволяет нам использовать определенные версии приложений.

**VirtualEnv** используется для создания виртуальных окружений для Python программ. Это необходимо для избежание конфликтов, позволяя установить одну версию библиотеки для одной программы, и другу для второй.

**virtualenv** - стандартный пакет для работы с виртуальным окружением. Используется вместе с **virtualenvwrapper** для более удобной работы.

## **Создание виртуального окружения**

Конечно же, вручную создавать всю описанную иерархию директорий и файлов не нужно — для этого есть специальный модуль venv.

В macOS и Windows этот модуль, как и pip, входит в поставку Python. На Ubuntu же его нужно установить отдельно командой

sudo apt install python3-venv

Проверим, что модуль установлен и пригоден к использованию:

python3 -m venv --help

usage: venv **[**-h**]** **[**--system-site-packages**]** **[**--symlinks | --copies**]** **[**--clear**]** **[**--upgrade**]** **[**--without-pip**]** **[**--prompt PROMPT] ENV\_DIR **[**ENV\_DIR ...]

Создаётся окружение командой python3 -m venv имя\_окружения.

## **Активация окружения**

При создании окружения в поддиректорию bin помещается сценарий оболочки, который на macOS и Ubuntu называется activate, а на Windows — activate.bat. Этот сценарий нужно выполнить:

* на macOS и Ubuntu вызвать команду

source first\_venv/bin/activate

* на Windows вызвать команду

C:\> first\_venv\Scripts\activate.bat

Заметьте, в Windows поддиректория с исполняемыми файлами называется не bin, а Scripts!

**Далее материалы из семинарского занятия (аудиторная работа по «Настройка виртуального окружения» в OneNote)**

*Создание виртуального окружения Python*

**#apt install virtualenv**

Это отдельная программа

в папке проекта test запускаем

#**ls –lah**

видим только один файл проекта

Создадим виртуальное окружение питона только для этого проекта для установки модулей только для этого проекта, что бы не засорять систему

**#Virtualenv vent-test**

Vent-test - имя проекта для читабельности и простоты восприятия

Появилась папка **#ls –lah**

Теперь активируем виртуальное окружение

**#source vent-test/bin/activate**

(vent-test) root@debian:~/test#

Поменялось системное приглашение

(vent-test) root@debian:~/test**# which python3**

/root/test/vent-test/bin/python3

Поменялось путь к питону

### Структура проекта на Python. Организация модулей. Файл зависимостей.

# **Модули в Python**

## **Что такое модуль в Python?**

Под **модулем в *Python*** понимается файл с расширением ***.py***. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке *Python*, но и на других языках (например *C*).

## **Как импортировать модули в Python?**

Самый простой способ импортировать модуль в *Python* это воспользоваться конструкцией:

***import имя\_модуля***

Импорт и использование модуля *math*, который содержит математические функции, будет выглядеть вот так.

>>> import math

>>> math.factorial(**5**)

120

За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова *import*:

***import имя\_модуля1, имя\_модуля2***

>>> import math, datetime

>>> math.cos(math.pi/**4**)

0.707106781186547

>>> datetime.date(**2017**, **3**, **21**)

datetime.date(2017, 3, 21)

Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом:

***import имя\_модуля as новое\_имя***

>>> import math as m

>>> m.sin(m.pi/**3**)

0.866025403784438

Используя любой из вышеперечисленных подходов, при вызове функции из импортированного модуля, вам всегда придется указывать имя модуля (или псевдоним). Для того, чтобы этого избежать делайте импорт через конструкцию *from … import…*

***from имя\_модуля import имя\_объекта***

>>> from math import cos

>>> cos(**3.14**)

0.999998731727539

При этом импортируется только конкретный объект (в нашем примере: функция *cos*), остальные функции недоступны, даже если при их вызове указать имя модуля.

**ФАЙЛЫ ЗАВИСИМОСТЕЙ**

Если вы предоставляете доступ к проекту другим, используете систему сборки или планируете скопировать проект в другое расположение, где нужно восстановить среду, то необходимо указать внешние пакеты, необходимые проекту.

Рекомендуется **использовать файл requirements.txt** (readthedocs.org), содержащий список команд для pip, который устанавливает необходимые версии зависимых пакетов. Наиболее распространенной является команда pip freeze > requirements.txt, которая записывает текущий список пакетов среды в **файл *requirements.txt*.**

Технически для отслеживания требований можно использовать любой файл (используя -r <full path to file> при установке пакета), но Visual Studio имеет встроенную поддержку *requirements.txt*:

· Если вы загрузили проект, содержащий файл *requirements.txt*, и хотите установить все указанные в нем пакеты, разверните узел **Среды Python** в **обозревателе решений**, щелкните правой кнопкой мыши узел среды и выберите **Установить из файла requirements.txt**:

· Если вы хотите установить зависимости в виртуальном окружении, сначала создайте и активируйте окружение, а затем воспользуйтесь командой **Установка из файла requirements.txt**.

· Если в среде установлены все необходимые пакеты, можно щелкнуть среду правой кнопкой мыши в **обозревателе решений** и выбрать **Создать файл requirements.txt**, чтобы создать необходимый файл. Если файл уже существует, отображается запрос с вариантами обновления:

**Файлы *requirements.txt*** содержат точные версии всех установленных пакетов, и вы можете использовать эти файлы, чтобы зафиксировать требования окружения. Такие версии позволяют легко воспроизвести ваше окружение на другом компьютере. Файлы с требованиями включают пакеты, даже если они были установлены с диапазоном версий, как зависимость от другого пакета или с установщиком, отличным от pip.

Если pip не удается установить пакет, указанный в файле *requirements.txt*, установка завершается сбоем.

### Системы контроля версий. Примеры, назначение, общие понятия.

**Системы контроля версий** (**Version Control System**, VCS) - представляет собой программное обеспечение, которое позволяет отслеживать изменения в документах, при необходимости производить их откат, определять, кто и когда внес исправления и т. п.

**Для чего нужен СКВ:**

· В свете усложнения сред разработки они помогают командам разработчиков работать быстрее и эффективнее.

· Программное обеспечение контроля версий отслеживает все вносимые в код изменения в специальной базе данных. При обнаружении ошибки разработчики могут вернуться назад и выполнить сравнение с более ранними версиями кода для исправления ошибок, сводя к минимуму проблемы для всех участников команды.

## **Централизованные и распределенные системы контроля версий**

Системы контроля версий можно разделить **на две группы:**

· распределенные

· централизованные

### Централизованные системы контроля версий

### 

**Централизованные системы контроля версий** представляют собой приложения типа клиент-сервер, когда репозиторий проекта существует в единственном экземпляре и хранится на сервере. Доступ к нему осуществлялся через специальное клиентское приложение. В качестве примеров таких программных продуктов можно привести *CVS*, *Subversion*.

### Распределенные системы контроля версий

### 

**Распределенные системы контроля версий** (*Distributed Version Control System, DVCS*) позволяют хранить репозиторий (его копию) у каждого разработчика, работающего с данной системой. При этом можно выделить центральный репозиторий (условно), в который будут отправляться изменения из локальных и, с ним же эти локальные репозитории будут синхронизироваться.

При работе с такой системой, пользователи периодически синхронизируют свои локальные репозитории с центральным и работают непосредственно со своей локальной копией. После внесения достаточного количества изменений в локальную копию они (изменения) отправляются на сервер. При этом сервер, чаще всего, выбирается условно, т.к. в большинстве *DVCS* нет такого понятия как “выделенный сервер с центральным репозиторием”.

Примеры таких СКВ: *Git, Mercurial*

### Общий алгоритм работы с СКВ Git. Инициализация репозитория, добавление файлов, коммиты.

**Git** — система управления версиями с распределенной архитектурой.

С помощью Git можно откатить проект до более старой версии, сравнить, проанализировать или слить свои изменения в репозиторий.

**Репозиторием** называют хранилище кода и историю его изменений (или коллекцией всех изменений, которые были совершены на протяжении всего времени после инициализации репозитория). Git работает локально и все репозитории хранятся в определенных папках на жестком диске. Так же репозитории можно хранить и в интернете (например, GitHub). В Git каждая рабочая копия кода сама по себе является репозиторием. Это позволяет всем разработчикам хранить историю изменений в полном объеме.

Каждая точка сохранения проекта, фиксирующая изменения, носит название **коммит (commit)**. У каждого commit-a есть hash (уникальный id) и комментарий. Из таких commit-ов собирается ветка. Ветка - это история изменений. У каждой ветки есть свое название. Репозиторий может содержать в себе несколько веток, которые создаются из других веток или вливаются в них.

#### **Общий алгоритм использования СКВ GIT:**

* Установить и настроить клиент git
* Создать репозиторий в папке с проектом
* Добавить необходимые файлы под СКВ
* Сделать первоначальный коммит
* Сделать логически завершенный участок работы
* Сделать новый коммит с поясняющим сообщением
* Повторить сколько нужно

Основные команды:

***git init*** – команда создает/инициализирует .git репозиторий в проекте.

***git add*** *имяФайла.расширение* – команда фиксирует изменения файла и добавляет файл в “staging area” (участок подготовки – секция, в которой файлы проходят подготовку к перемещению в репозиторий).

***git add*** *.* – добавление всех файлов из директории проекта в staging area.

***git status*** – отображение ранее добавленных файлов в staging area и файлов, которые были изменены и ждут перемещения в staging area.

***git reset*** *имяФайла.расширение* – удаление выбранного файла из staging area.

***git commit -m*** *"Описание коммита"* – фиксирование файлов из staging area в локальный репозиторий. В кавычки следует вставить краткое описание изменений для конкретного коммита.

### Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток. Разрешение конфликтов слияния.

**Назначение веток:**

* Ветвление происходит, когда мы произвели два разных изменения, основываясь на одном и том же состоянии.
* Часто происходит при командной работе.
* Git позволяет работать с ветками комфортно и, при необходимости, объединять их.
* **Ветвление** - это способ изолировать одни изменения от других. Например, разработка одной фичи независимо от хода работ над другой.

**Создание веток:**

**Ветки** - это специальные указатели на коммиты, которые перемещаются с каждым новым коммитом.

Создать новую ветку можно командой **git branch <name>**

Команда git branch выводит все ветки.

Ветка по умолчанию по соглашению называется master.

Git хранит специальный указатель HEAD - он показывает, на какой ветке вы сейчас находитесь. Чтобы перейти на существующую ветку, вам надо выполнить команду **git checkout.** Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом. Теперь коммиты, которые мы будем делать, будут относиться к новой ветке.

**Переключение на другую ветку:**

Чтобы перейти на существующую ветку, вам надо выполнить команду **git checkout.**

В любой момент можно перейти в другую ветку и начать коммитить в нее. При переходе в другую ветку состояние рабочей директории приводится в соответствие с состоянием последнего коммита в этой ветке. Поэтому все изменения будут основываться на том состоянии.

Переход между ветками откатывает изменения, которые вы делали в другой ветке, но не удаляет их. Все фиксированные изменения сохраняются в истории веток.

При использовании веток история коммитов уже не будет линейной.

**Объединение веток:**

Две ветки можно объединить (слить). Это значит, что мы применяем к файлам изменения, которые произошли и в одной ветке и во второй.

Команда **git merge <branch>** сливает указанную ветку в текущую. Это значит, что создастся новый коммит в текущей ветке, учитывающий изменения в указанной.

* Коммиты слияния имеют двух родителей.
* Git автоматически определяет наилучшего общего предка для слияния веток.
* После слияния ветка не уничтожается и над ней можно продолжать работать.

**Разрешение конфликтов слияния**

Если вы изменили одну и ту же часть файла по-разному в двух ветках, которые собираетесь слить, Git не сможет сделать это чисто. **Такая ситуация называется конфликтом слияния.**

В таком случае, git приостанавливает слияние до тех пор, пока вы не разрешите конфликт.

Посмотреть, какие файлы не прошли слияние можно командой **git status.**

Содержимое спорных файлов в рабочей директории будет отражать оба конфликтующих варианта текста.Чтобы разрешить конфликт, вы должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению.После этого необходимо выполнить добавление этих файлов к индексу и коммит для завершения слияния веток.

### Работа с удаленными репозиториями. Клонирование и форк репозиториев. Отправка и получение изменений в удаленный репозиторий.

**Работа с удаленными репозиториями:**

При командной разработке приходится синхронизировать работу нескольких программистов в одном месте. Даже одному разработчику иногда приходится работать и вносить изменения в проект из разных мест. Для этого в системах контроля версий предусмотрена возможность связать локальный репозиторий с другим - удаленным.

При связывании репозиториев возникает возможность отправить изменения, сделанные в локальном репозитории в удаленный и наоборот - скачать изменения из удаленного репозитория в локальный.

Чтобы просмотреть, какие удалённые серверы у вас уже настроены, следует выполнить команду **git remote.** Она перечисляет список имён-сокращений для всех уже указанных удалённых дескрипторов. Если вы склонировали ваш репозиторий, у вас должен отобразиться, по крайней мере, origin — это имя по умолчанию, которое Git присваивает серверу, с которого вы склонировали.

**Клонирование репозиториев:**

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду **git clone**.  
Вместо того, чтобы просто получить рабочую копию, **Git получает копию практически всех данных, которые есть на сервере.**

Эта команда создаёт директорию “libgit2”, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования.

В Git реализовано несколько транспортных протоколов, которые вы можете использовать.

**Форк репозиториев:**

Если вы хотите вносить свой вклад в уже существующие проекты, в которых у нас нет прав на внесения изменений путем отправки (push) изменений, **вы можете создать свое собственное ответвление (“fork”) проекта**.

Это означает, что GitHub создаст вашу собственную копию проекта, данная копия будет находиться в вашем пространстве имен и вы сможете легко делать изменения путем отправки (push) изменений.

Таким образом, проекты не обеспокоены тем, чтобы пользователи, которые хотели бы выступать в роли соавторов, имели право на внесение изменений путем их отправки (push). Люди просто могут создавать свои собственные ветвления (fork), вносить туда изменения, а затем отправлять свои внесенные изменения в оригинальный репозиторий проекта путем создания запроса на принятие изменений (Pull Request)

Для того, чтобы создать ответвление проекта (fork), зайдите на страницу проекта и нажмите кнопку “Cоздать ответвление” (“Fork”), которая расположена в правом верхнем углу.

**Отправка изменений в удаленный репозиторий:**

$ git push origin master

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий.

Команда для этого действия: **git push [удал. сервер] [ветка].**

Если вы и кто-то ещё одновременно клонируете, затем он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонён.

**Получение изменений в удаленный репозиторий:**

Можно использовать команду **git pull** чтобы автоматически получить изменения из удалённой ветви и слить их со своей текущей ветвью.

Выполнение git pull, как правило, извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете.

### Современные методологии работы с Git в командном проекте. GitFlow.

Распределенная модель git позволяет гибко взаимодействовать разработчикам в командном проекте.

В централизованных системах все разработчики являются узлами сети, более или менее одинаково работающими на центральном хабе.

Это означает, что если два разработчика склонируют репозиторий и каждый внесет изменения, то первый из них сможет отправить свои изменения в репозиторий без проблем. **Второй разработчик должен слить изменения**, сделанные первым разработчиком, чтобы избежать их перезаписи во время отправки на сервер.

Однако, в Git'е **каждый разработчик потенциально является и узлом, и хабом**. То есть каждый разработчик может как вносить код в другие репозитории, так и содержать публичный репозиторий, на основе которого работают другие разработчики, и в который они вносят свои изменения.

**Дополнительно**

**Ветви:**

* Старайтесь давать ветвям в git короткие, описательные названия:
* Используйте разнообразные идентификаторы задач (например, задача, созданная в сервисе Github).
* Если в названии несколько слов, для разделения используйте дефис «-».
* Когда несколько человек одновременно работают над одним функционалом, можно создать отдельную ветку на каждого разработчика и общую ветвь, в которой хранятся все изменения, используя соответствующее наименование.   Все изменения из личных веток будут сливаться в общую, которая по завершению функционала будет, в свою очередь, добавлена в master. Если нет необходимости хранить изменения, сделанные в ветке, то после слияния ее необходимо удалить из удаленного репозитория.

**Коммиты:**

* Каждый коммит логически должен представлять собой одно законченное изменение. Не стоит включать несколько таких изменений в один коммит. И наоборот, не стоит разбивать одно логическое изменение на несколько коммитов.
* Постоянно сохраняйте свои изменения. Небольшие атомарные коммиты легче понять и отменить, если что-то пойдет не так.
* Коммиты должны идти в логическом порядке. Если коммит Х зависит от изменений, сделанных в коммите Y, то эти изменения нужно сохранить перед изменениями коммита Х.

**Сообщения к коммитам:**

* Для написания сообщения к коммиту лучше использовать текстовый редактор, а не консоль.  Сохранение изменений в консоли заставляет писать сообщения длинной не более одной строки, что в итоги выливается в неинформативные и неоднозначные сообщения к коммиту.
* Сообщение должно быть описательным и лаконичным. В идеале, оно не должно быть более 50 символов и начинаться с большой буквы.
* После названия необходимо дать более полное описание изменений, сделанных в коммите. При этом название и описание надо разделить пустой строкой. Описание должно быть ограничено 72 символами, кроме того, в нем нужно пояснить, для чего именно были внесены эти изменения, как это изменение решает поставленную задачу и какие побочные эффекты возможны при этом. Кроме того, в описание необходимо включить связанные с коммитом ресурсы (например, ссылку на соответствующую задачу в системе для отслеживания ошибок).
* Иногда первая строка коммита трактуется как тема электронного письма, а остальное - как его тело. Пустая линия, которая разделяет "тему" от "тела" письма необходима, иначе некоторые команды воспримут их как одну.
* Можно использовать ненумерованные списки с точкой, дефисом либо звездочкой в качестве маркера, отделяя при этом пункты пустыми строками.
* Если коммит А зависит от коммита В, то факт этой зависимости должен быть записан в сообщении к коммиту А. Для ссылки на коммиты используйте их хеш. Аналогично, если в коммите А был исправлен баг, допущенный в коммите В, то этот факт также должен быть упомянут в сообщении к коммиту А.

**Слияние изменений:**

* Не переписывайте историю репозитория, если она уже опубликована. По ней можно понять, что произошло на самом деле. Изменение истории репозитория может обернуться большими проблемами для всех, кто с ним работает.
* Не нужно переписывать историю веток.
* Старайтесь поддерживать историю коммитов ясной и простой.
* Если у вас в ветке больше одного коммита, не стоит делать автоматическое слияние.

**Для чего нужен GitFlow?**

#### Методология работы с git

#### Стандартизировать рабочий процесс с использованием веток

#### Вводит некоторые полезные и распространенные понятия

#### Регламентирует правила слияния веток

#### Сводит к минимуму риск выложить нестабильный код в мастер

#### Хорошо использовать с непрерывной интеграцией

#### Есть инструментальные решения.

Git-flow — альтернативная модель ветвления Git, в которой используются функциональные ветки и несколько основных веток. Она предполагает выстраивание строгой модели ветвления вокруг релиза проекта, которая дает надежную схему управления крупными проектами. Gitflow подходит для проектов, которые имеют спланированный цикл релиза.

В соответствии с этой моделью разработчики создают функциональную ветку и откладывают ее слияние с главной магистральной веткой до завершения работы над функцией. Такие долгосрочные функциональные ветки требуют тесного взаимодействия разработчиков при слиянии и создают повышенный риск отклонения от магистральной ветки. В них также могут присутствовать конфликтующие обновления.

Gitflow — это лишь методика работы с Git; в ней определяется, какие виды веток необходимы проекту и как выполнять слияние между ними. Набор инструментов git-flow представляет собой отдельную утилиту командной строки, которая требует установки. Пакеты команд git-flow доступны для многих операционных систем.

**Ветки в GitFlow**

* В GitFlow есть два две основные ветки: master и develop.  
  В ветке master содержится ровно тот же код, что и в рабочей (читай, продакт) версии проекта. А вся работа делается в ветке develop.
* Во время работы на основе develop создаются так называемые **feature-ветки**. Их может быть неограниченное количество.
* Далее, у нас есть ветка **release**, которая используется для подготовки к новому релизу проекта.
* Наконец, есть ветка **hotfix**, которая служит для срочного исправления багов, найденных, например, на продакте.

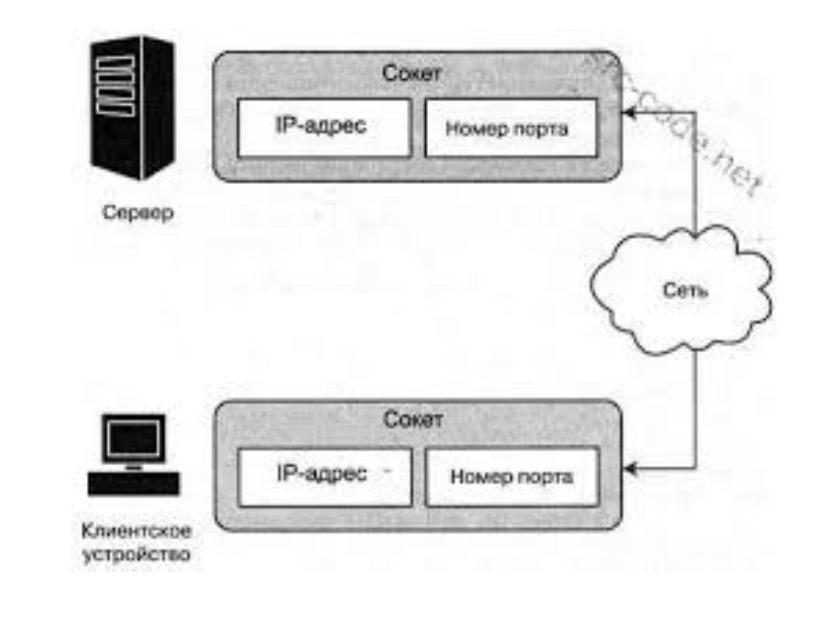
**Основа работы с гитфлоу** - хранить в master только релизный код. Там не допускается рабочий процесс, потенциально приводящий к ошибкам.

### Понятие сетевого сокета. Применение, виды, схема взаимодействия.

**Со́кет** — название программного интерфейса для обеспечения обмена данными между процессами. Сокеты не требуют специального программного обеспечения.

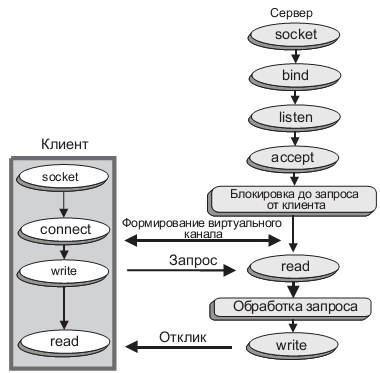
Пара IP-адрес и порт характеризуют сокет (гнездо) - начальную или конечную точку сетевой коммуникации. Для создания соединения нужно два сокета: один на локальной машине, а другой - на удаленной.

**Схема соединения сокетов:**



**Алгоритм установки соединения:**

1. Сначала должен быть создан серверный сокет.
2. Он привязывается к определенному номеру порта и начинает ждать соединения.
3. Затем создается клиентский сокет.
4. Он присоединяется к серверному по адресу хоста и номеру порта
5. После этого устанавливается двунаправленное соединение
6. Используется механизм системных вызовов операционной системы



**Виды сокетов:**

1. **Потоковые сокеты (stream socket)**

Потоковый сокет — это сокет с установленным соединением, состоящий из потока байтов, который может быть двунаправленным, т, е. через эту конечную точку приложение может и передавать, и получать данные.

1. **Дейтаграммные сокеты (datagram socket)**

Дейтаграммные сокеты иногда называют сокетами без организации соединений, т. е. никакого явного соединения между ними не устанавливается — сообщение отправляется указанному сокету и, соответственно, может получаться от указанного сокета.

**3. Сырые сокеты (raw socket)**

Главная цель использования сырых сокетов состоит в обходе механизма, с помощью которого компьютер обрабатывает TCP/IP. Это достигается обеспечением специальной реализации стека TCP/IP, замещающей механизм, предоставленный стеком TCP/IP в ядре — пакет непосредственно передается приложению и, следовательно, обрабатывается гораздо эффективнее, чем при проходе через главный стек протоколов клиента.

По определению, сырой сокет — это сокет, который принимает пакеты, обходит уровни TCP и UDP в стеке TCP/IP и отправляет их непосредственно приложению.

**4. SOCK\_SEQPACKET**

Тип сокета SOCK\_SEQPACKET подобен типу SOCK\_STREAM и также ориентирован на соединение. Обеспечивает работу последовательного двустороннего канала для передачи датаграмм с поддержкой соединений; датаграммы имеют ограниченную длину; от получателя требуется за один раз прочитать целый пакет.

**5. SOCK\_RDM**

Обеспечивает надежную доставку датаграмм без гарантии, что они будут расположены по порядку.

### Блокирующие операции при обмене через сокеты. Возможные ошибки. Таймауты.

Закрытие сокета

Строго говоря, сначала необходимо использовать вызов объекта сокета shutdown(), прежде чем закрыть его командой Socket.close(). Вызов Socket.shutdown() - это предупреждение для сокета на другом конце. В зависимости от аргумента, который передавать, это может означать *"Я больше не буду отправлять, но я все равно буду слушать"* или *"Я не слушаю, мне по барабану"*. Разработчики библиотек сокетов настолько привыкли к тому, что программисты пренебрегают этим элементом этикета, что у некоторых, обычный вызов Socket.close() - означает последовательность вызовов: Socket.shutdown(); Socket.close(). Поэтому в большинстве ситуаций явный вызов Socket.shutdown() не требуется.

Один из способов эффективного использования shutdown(), это обмен данными, подобный HTTP. Клиент отправляет запрос и затем завершает работу вызовом Socket.shutdown(1). Это сообщает серверу: *"Этот клиент завершил отправку, но все еще может получать"*. Сервер читая запрос, в конце получает 0 байтов, это сигнализирует о том, что от клиента весь запрос получен и надо готовить и отправлять ответ. Если отправка ответа завершилась успешно, то клиент действительно слушал и получил все отправленные данные.

Python делает еще один шаг к автоматическому завершению соединения, это постоянный мониторинг открытых сокетов сборщиком мусора. Сборщик мусора автоматически закрывает соединение, если это необходимо. Но полагаться на это - очень плохая привычка. Если сокет просто исчезнет без закрытия, то сокет на другом конце может зависнуть бесконечно думая, что сервер просто медленно работает и когда закончит, то закроет сокет.

### Когда умирают сокеты.

Худшее в использовании блокирующих сокетов - это то, что происходит, когда одна из сторон соединения резко падает (без закрытия). Сокет скорее всего зависнет. TCP - надежный протокол и он будет долго ждать, прежде чем отказаться от соединения. Если использовать потоки, то весь поток практически умрет. С этим ничего не поделаешь и если не делать глупостей, таких как держать блокировку при выполнении чтения, то поток не съест много ресурсов.

Не пытайтесь убить поток - одна из причин того, что потоки более эффективны, чем процессы, заключается в том, что у них нет накладных расходов, связанных с автоматическим повторным использованием ресурсов. Другими словами, если удастся убить поток, то вся программа, скорее всего, упадет.

### Транспортные протоколы TCP и UDP. Принципы работы, сравнение.

**TCP** – транспортный протокол передачи данных в сетях TCP/IP, предварительно устанавливающий соединение с сетью.

**UDP** – транспортный протокол, передающий сообщения-датаграммы без необходимости установки соединения в IP-сети.

Оба протокола работают на транспортном уровне модели OSI или TCP/IP



**Отличия**:

1. TCP гарантирует доставку пакетов данных в неизменных виде, последовательности и без потерь, UDP ничего не гарантирует.

2. TCP нумерует пакеты при передаче, а UDP нет

3. TCP работает в дуплексном режиме, в одном пакете можно отправлять информацию и подтверждать получение предыдущего пакета.

4. TCP требует заранее установленного соединения, UDP соединения не требует, у него это просто поток данных.

5. UDP обеспечивает более высокую скорость передачи данных.

6. TCP надежнее и осуществляет контроль над процессом обмена данными.

7. UDP предпочтительнее для программ, воспроизводящих потоковое видео, сетевых игр.

8. UPD не содержит функций восстановления данных

### Клиент-серверное взаимодействие.

Данная концепция взаимодействия была разработана в первую очередь для того, чтобы разделить нагрузку между участниками процесса обмена информацией, а также, чтобы разделить программный код поставщика и заказчика.

· Сначала должен быть создан серверный сокет.

· Он привязывается к определенному номеру порта и начинает ждать соединения.

· Затем создается клиентский сокет.

· Он присоединяется к серверному по адресу хоста и номеру порта

· После этого устанавливается двунаправленное соединение

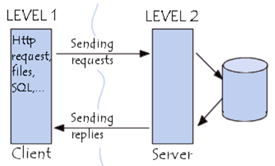
· Используется механизм системных вызовов операционной системы

**Архитектура клиент-сервер определяет лишь общие принципы взаимодействия между компьютерами**, детали взаимодействия определяют различные протоколы. При этом взаимодействие всегда начинает клиент, а правила, по которым происходит взаимодействие описывает протокол.

Существует два вида архитектуры взаимодействия клиент-сервер: первый получил название **двухзвенная архитектура клиент-серверного взаимодействия**, второй – **многоуровневая архитектура клиент-сервер.**

**Двухзвенная архитектура**

Принцип работы двухуровневой архитектуры взаимодействия клиент-сервер заключается в том, что обработка запроса происходит на одной машине без использования сторонних ресурсов. Двухзвенная архитектура предъявляет жесткие требования к производительности сервера, но в то же время является очень надежной.

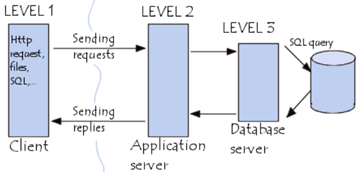


Есть клиент (1 уровень), который позволяет человеку сделать запрос, и есть сервер, который обрабатывает запрос клиента.

**Многоуровневая архитектура**

Как пример можно взять СУБД.

Суть многоуровневой архитектуры заключается в том, что запрос клиента обрабатывается сразу несколькими серверами. Такой подход позволяет значительно снизить нагрузку на сервер из-за того, что происходит распределение операций, но в то же самое время данный подход не такой надежный, как двухзвенная архитектура.



Если говорить в контексте систем управления базами данных, то первый уровень – это клиент, который позволяет нам писать различные [SQL запросы](https://zametkinapolyah.ru/zametki-o-mysql/chast-5-sql-zaprosy-klyuchevye-slova-komandy-predlozheniya-i-sintaksis-yazyka-sql.html) к [базе данных](https://zametkinapolyah.ru/zametki-o-mysql/bazy-dannyx-vidy-i-tipy-baz-dannyx-struktura-relyacionnyx-baz-dannyx-proektirovanie-baz-dannyx-setevye-i-ierarxicheskie-bazy-dannyx.html). Второй уровень – это движок СУБД, который интерпретирует запросы и реализует взаимодействие между клиентом и файловой системой, а третий уровень – это хранилище данных.

**Преимуществом** модели взаимодействия клиент-сервер является то, что программный код клиентского приложения и серверного разделен**.**

К **недостаткам** модели взаимодействия клиент-сервер можно отнести то, что стоимость серверного оборудования значительно выше клиентского. Сервер должен обслуживать специально обученный и подготовленный человек.

### Реализация сокетов в языке Python. Модуль socket.

**Сокет** — это программный интерфейс для обеспечения информационного обмена между процессами.Низкоуровневый сетевой интерфейс.

Существуют клиентские и серверные сокеты. Серверный сокет прослушивает определенный порт, а клиентский подключается к серверу. После того, как было установлено соединение начинается обмен данными.

Сервер

В Python для работы с сокетами используется модуль socket:

**import** socket

Прежде всего нам необходимо создать сокет:

sock = socket.socket()

Хост и порт

Насчет хоста — мы оставим строку пустой, чтобы сервер был доступен для всех интерфейсов. А порт возьмем любой от нуля до 65535. Следует отметить, что в большинстве операционных систем прослушивание портов с номерами 0 — 1023 требует особых привилегий. Выберем 9090. Теперь свяжем наш сокет с данными хостом и портом с помощью метода bind, которому передается кортеж, первый элемент которого — хост, а второй — порт:

sock.bind(('', 9090))

Принятие соединения

С помощью метода listen запустим режим прослушивания. Метод принимает один аргумент — максимальное количество подключений в очереди. Установим его в единицу:

sock.listen(1)

Наконец-то, мы можем принять подключение с помощью метода accept, который возвращает кортеж с двумя элементами: новый сокет и адрес клиента. Именно этот сокет и будет использоваться для приема и посылке клиенту данных.

conn, addr = sock.accept()

Теперь мы установили с клиентом связь и можем с ним «общаться». Т.к. мы не можем точно знать, что и в каких объемах клиент нам пошлет, то мы будем получать данные от него небольшими порциями. Чтобы получить данные нужно воспользоваться методом recv, который в качестве аргумента принимает количество байт для чтения. Мы будем читать порциями по 1024 байт:

**while** True:

data = conn.recv(1024)

**if** **not** data:

**break**

conn.send(data.upper())

Для наглядности что-то сделаем с полученными данными и отправим их обратно клиенту. Например, с помощью метода upper у строк вернем клиенту строку в верхнем регистре.

Теперь можно и закрыть соединение:

conn.close()

Собственно сервер готов. Он принимает соединение, принимает от клиента данные, возвращает их в виде строки в верхнем регистре и закрывает соединение

#### **Клиент**

Клиентское приложение проще — нам нужно создать сокет, подключиться к серверу послать ему данные, принять данные и закрыть соединение. Все это делается так:

**import** socket

sock = socket.socket()

sock.connect(('localhost', 9090))

sock.send('hello, world!')

data = sock.recv(1024)

sock.close()

**print** data

Метод connect, с помощью него подключаемся к серверу. Дальше читаем 1024 байт данных и закрываем сокет.

Отдельно вынесен код:

| **Сервер** | **Клиент** |
| --- | --- |
| import socket  sock = socket.socket()  sock.bind(('', 9090))  sock.listen(1)  conn, addr = sock.accept()  print('connected:', addr)  while True:  data = conn.recv(1024)  if not data:  break  conn.send(data.upper())  conn.close() | import socket    sock = socket.socket()  sock.connect(('localhost', 9090))  sock.send('hello, world!')    data = sock.recv(1024)  sock.close()    print(data) |

### Понятие программного потока. Процессы и потоки.

**Процесс** — экземпляр программы во время выполнения, независимый объект, которому выделены системные ресурсы (например, процессорное время и память).

· Каждый процесс выполняется в отдельном адресном пространстве: один процесс не может получить доступ к переменным и структурам данных другого.

· Если процесс хочет получить доступ к чужим ресурсам, необходимо использовать межпроцессное взаимодействие. Это могут быть конвейеры, файлы, каналы связи между компьютерами и многое другое.

**Поток** — определенный способ выполнения процесса. Когда один поток изменяет ресурс процесса, это изменение сразу же становится видно другим потокам этого процесса.

· Поток использует то же самое пространства стека, что и процесс, а множество потоков совместно используют данные своих состояний.

· Как правило, каждый поток может работать (читать и писать) с одной и той же областью памяти, в отличие от процессов, которые не могут просто так получить доступ к памяти другого процесса.

· У каждого потока есть собственные регистры и собственный стек, но другие потоки могут их использовать.

**Простое создание потока**

За создание, управление и мониторинг потоков отвечает класс *Thread* из модуля *threading*.

target=proc - вызываемый объект (функция), который будет вызываться [методом Thread.run()](https://docs-python.ru/standart-library/modul-threading-python/klass-thread-modulja-threading/#Thread.run).

Метод Thread.run() представляет активность потока, вызывает вызываемый объект (функцию)

name= - имя потока.

import threading

def proc():

print("Процесс")

p1 = threading.Thread(target=proc, name="t1")

p2 = threading.Thread(target=proc, name="t2")

p1.start()

p2.start()

В результате программа напечатает:

Процесс

Процесс

То же самое можно проделать через наследование от класса threading.Thread с определением собственного конструктора и метода run():

import threading

class T(threading.Thread):

def \_\_init\_\_(self, n):

threading.Thread.\_\_init\_\_(self, name="t" + n)

self.n = n

def run(self):

print "Процесс", self.n

p1 = T("1")

p2 = T("2")

p1.start()

p2.start()

### Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.

**Синхронное** выполнение программы подразумевает последовательное выполнение операций.

**Асинхронное** – предполагает возможность независимого выполнения задач.

**Конкурентность** предполагает выполнение нескольких задач одним исполнителем. Из примера с готовкой: один человек варит картошку и прибирается, при этом, в процессе, он может переключаться: немного прибрался, пошел помешал-посмотрел на картошку, и делает он это до тех пор, пока все не будет готово.

**Параллельность** предполагает параллельное выполнение задач разными исполнителями: один человек занимается готовкой, другой приборкой.

**Многозадачность** (асинхронность, конкретность, параллельность):

| **Преимущества** | **Недостатки** |
| --- | --- |
| Выигрыш по скорости  Повышение интерактивности  Выполнение операций в фоне | Некоторые задачи в принципе являются последовательными  Сложность проектирования  Потокобезопасность  Сложность отладки  Непредсказуемое переключение контекста  Накладные расходы  Вероятность взаимоблокировки  Голодание задач |

[**asyncio**](https://docs.python.org/3/library/asyncio.html#module-asyncio) **— Асинхронный ввод-вывод.**

· Введен в Python с версии 3,5

· Позволяет писать конкурентные корутины

· Включает собственную реализацию блокирующих операций

· Позволяет писать ясный код

· Цикл выполнения контролируется интерпретатором

· Позволяет явно прописывать места переключения потока выполнения

· High-level и low-level API

**Основные понятия**

· корутины — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий

· task в asyncio – это объект, который оборачивает coroutine, предоставляя методы для контроля ее выполнения и запроса ее статуса. task может быть создан с помощью asyncio.create\_task() или asyncio.gather().

· В asyncio event loop управляет планированием и передачей ожидаемых объектов. Каждая программа asyncio имеет как минимум один event loop. Можно иметь несколько циклов, но в Python 3.7 настоятельно рекомендуется использовать только один.

**Пример**:

async def g(x):

yield x # Это асинхронный генератор

### Блокирующие и неблокирующие операции.

Есть две модели программирования сокетов – **блокирующий** и **неблокирующий**. Время от времени они также именуются как – синхронный (blocking) и асинхронный (non-blocking).

В случае **блокирующих** сокетов при попытке прочитать (и записать) данные функция чтения будет ждать до тех пор, пока не прочитает хотя бы один байт или произойдет разрыв соединения или придет сигнал. В случае **неблокирующих** сокетов функция чтения проверяет, есть ли данные в буфере, и если есть - сразу возвращает, если нет, то она не ждет и также сразу возвращает, что прочитано 0 байт.

*Плюсы блокирующего режима*

1. **Проще программировать**. Весь пользовательский код может находиться в одном месте и выполняться в естественном, поочередном порядке.

2. **Кросс-платформенность** –Unix употребляет блокирующие сокеты, то переносимый код легче писать.

3. **Удобнее работать с потоками** - Так как у блокирующих сокетов последовательность приобретена по наследственности, потому их просто применять в потоках.

4. **Независимость от сообщений** – неблокирующие сокеты зависят от системы оконных сообщений. Когда, в конце концов, употребляются потоки, то создается отдельная очередь сообщений. Но когда потоки не употребляются, то узким местом становится обработка множества соединений.

*Недочеты блокирующего режима*

1. **Пользовательский интерфейс замораживается в клиентах** - Вызов блокирующего сокета не возвращает управления, пока не выполнит свою задачку. Когда подобные вызовы делаются в главном кодовом потоке, то приложение, наконец, замораживает пользовательский интерфейс. Замораживание происходит, так как сообщения обновления не обрабатываются до окончания вызова блокирующего сокета.

*Плюсы неблокирующего режима*

1. **Пользовательский интерфейс не замораживается** – так как пользовательский код обрабатывает оконные сообщения, то имеет контроль и над сокетными сообщениями.

2. **Многозадачность без использования потоков** – употребляется единственный кодовый поток для обработки множества сокетов.

3. **Чрезвычайно малая перегрузка при множестве сокетов** – так как множество сокетов могут обрабатываться без потоков, то перегрузка на память и процессор существенно ниже.

*Недочеты неблокирующего режима*

1. **Более сложное программирование** – неблокирующие сокеты требуют использования опроса либо обработки событий. При использовании обработчиков событий, код требует отслеживание состояния. Это значит большее количество ошибок

### Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.

Процессы можно классифицировать как те, которые *ограничены скоростью ввода-вывода* (*I/O-bound*), и те, которые *ограничены скоростью процессора* (*processor-bound*). К первому типу относятся процессы, которые большую часть своего времени выполнения тратят на отправку запросов на ввод-вывод информации и на ожидание ответов на эти запросы.

Процессы, ограниченные скоростью процессора, наоборот, большую часть времени исполняют программный код. Такие процессы обычно выполняются до того момента, пока они не будут вытеснены, так как эти процессы не блокируются в ожидании на запросы ввода-вывода.

Стратегия планирования процессов, ограниченных скоростью процессора, поэтому предполагает, что такие процессы должны выполняться реже, но более продолжительный период времени.

### Особенности реализации многопоточности в Python. Модуль threading.

Многопоточность — это выполнение программы сразу в нескольких потоках, которые выполняют её функции одновременно.

В чём **преимущества тогда модуля Threading**:

* Простота использования.
* Проще передавать данные из потока в основную программу. Вообще можно даже использовать глобальные переменные. Но при этом надо правильно проектировать программу, чтобы не было ошибок, связанных с «Состоянием гонки», которые мы рассмотрим ниже.

Threading – это стандартный модуль, который поставляется вместе с интерпретатором. Программисту не нужно устанавливать его, достаточно просто подключить модуль с помощью команды:

import threading

Работать с потоками можно, создавая экземпляры [класса](https://all-python.ru/osnovy/klassy.html) Thread. Чтобы создать отдельный, поток нужно создать экземпляр класса и применить к нему метод start(). Вот пример:

import threading

def myfunc(a, b):

print('сумма :',a + b)

thr1 = threading.Thread(target = myfunc, args = (1, 2)).start()

print('основной поток')

основной поток

сумма : 3

Здесь мы [функцию](https://all-python.ru/osnovy/funktsii.html) mydef запустили в отдельном потоке. В качестве аргументов функции передали числа 1 и 2.

### **threading.Thread()**

Эта конструкция позволяет создать новый поток, создав экземпляр класса Thread. Вот как выглядят её аргументы:

**Она принимает аргументы**:

threading.Thread(group=None, target=None, name=None, args=(),

kwargs={}, \*, daemon=None)

**Рассмотрим их подробнее**:

* group. Имеет значение None, зарезервирована для будущего расширения при реализации класса ThreadGroup.
* target. Это функция, которая выполняется в потоке с помощью метода run(), если передано значение None, ничего не вызывается.
* name. Это имя потока, по умолчанию оно принимает значение «Thread-X», где X – десятичное число. Программист может задать имя вручную.
* args. Это [кортеж](https://all-python.ru/osnovy/kortezh.html), в котором хранятся аргументы, передаваемые в вызываемую функцию.
* kwargs. Это [словарь](https://all-python.ru/osnovy/slovari.html), в котором хранятся аргументы, передаваемые в функцию.
* daemon. Это параметр, который устанавливает, является ли поток демоническим. По умолчанию имеет значение None, тогда свойство daemonic наследуется от текущего потока. Программист может самостоятельно установить значение параметра.

<https://all-python.ru/osnovy/threading.html#podklyuchenie-biblioteki-threading>

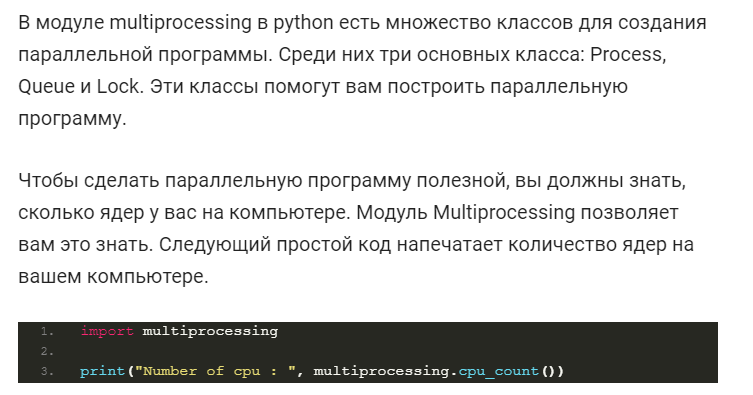
### Особенности организации многопроцессорной программы в Python. Модуль multiprocessing.

Многопроцессорность в Python – это способность системы запускать несколько процессов параллельно. Проще говоря, многопроцессорность использует два или более ЦП в одной компьютерной системе. Этот метод также может распределять задачи между процессами.

Вот почему важна необходимость концепции многопроцессорности:

* Многопроцессорность можно представить как компьютер с более чем одним центральным процессором.
* Многоядерный процессор относится к одному вычислительному компоненту с двумя или более независимыми блоками.

В многопроцессорном режиме ЦП может назначать несколько задач, каждая из которых имеет свой собственный процессор.



[multiprocessing](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/multiprocessing.html#module-multiprocessing) - это пакет, который поддерживает порождение процессов с использованием API, похожим на модуль [threading](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/threading.html#module-threading). Пакет [multiprocessing](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/multiprocessing.html#module-multiprocessing) предлагает как локальный, так и удаленный параллелизм, эффективно обходя [Глобальную блокировку интерпретатора](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/glossary.html#term-52) с использованием подпроцессов вместо потоков. Также модуль [multiprocessing](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/multiprocessing.html#module-multiprocessing) позволяет программисту полностью использовать несколько процессоров на данном компьютере.

В зависимости от платформы, [multiprocessing](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/multiprocessing.html#module-multiprocessing) поддерживает три способа запуска процесса. Далее перечислены *методы запуска*.

*spawn*

Родительский процесс запускает новый процесс интерпретатора Python. Дочерний процесс унаследует только те ресурсы, которые необходимы для выполнения метода [run()](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/multiprocessing.html#multiprocessing.Process.run) объекта процесса. В частности, ненужные файловые дескрипторы и дескрипторы родительского процесса не будут унаследованы. Запуск процесса с использованием этого метода довольно медленный по сравнению с использованием *fork* или *forkserver*.

Доступно в Unix и Windows. Значение по умолчанию для Windows и macOS.

*fork*

Родительский процесс использует [os.fork()](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/os.html#os.fork) для форсирования Python интерпретатор. Дочерний процесс, когда он начинается, фактически идентичен родительскому процессу. Все ресурсы родителя наследуются дочерним процессом. Следует отметить, что безопасная форсировка многопоточного процесса является проблематичной.

Доступно только в Unix. Значение по умолчанию в Unix.

*forkserver*

Когда программа начинает и выбирает метод начала *forkserver*, процесс сервера, начат. С этого момента всякий раз, когда требуется новый процесс, родительский процесс подключается к серверу и запрашивает, чтобы он форсировал новый процесс. Процесс сервера вилки является однопотоковым, поэтому он безопасен для использования [os.fork()](http://grep.cs.msu.ru/python3.8_RU/digitology.tech/docs/python_3/library/os.html#os.fork). Ненужные ресурсы не наследуются.

Доступный на платформах Unix, которые поддерживают мимолетный файл дескрипторы по Unix пайпы.

### Асинхронное программирование в Python. Использование asyncio.

Асинхронное программирование — это особенность современных языков программирования, которая позволяет выполнять операции, не дожидаясь их завершения.

Asyncio – это библиотека Python, которая используется для запуска параллельного кода с использованием async / wait. Это основа для асинхронной среды Python, которая предлагает библиотеки подключений, сетевые и веб-серверы, распределенные очереди задач базы данных, высокую производительность и т. д.

асинхронность предоставляет способ открытия тысячи соединений одновременно и переключения между ними. По сути, появляется возможность открыть соединение и переходить к следующему, ожидая ответа от первого. Так продолжается до тех пор, пока все не вернут результат.

Асинхронность больше всего подходит для таких сценариев:

* Программа выполняется слишком долго.
* Причина задержки — не вычисления, а ожидания ввода или вывода.
* Задачи, которые включают несколько одновременных операций ввода и вывода.

Это могут быть:

* Парсеры,
* Сетевые сервисы.

Ключевое слово *async*, используемое перед оператором *def*, определяет новую сопрограмму. Выполнение функции сопрограммы может быть приостановлено и возобновлено в строго определенных обстоятельствах.

**async** **def** **async\_hello**():

print("hello, world!")

Функции, определенные с помощью ключевого слова *async*, являются специальными. При вызове они не выполняют код внутри, а вместо этого возвращают объект сопрограммы:

>>>> async def async\_hello():

... print("hello, world!")

...

>>> async\_hello()

<coroutine object async\_hello at 0x1014129e8>

### Параллельное программирование. Достоинства и недостатки.

Параллельное программирование - это техника программирования, которая использует преимущества многоядерных или многопроцессорных компьютеров и является подмножеством более широкого понятия многопоточности.

Основные преимущества и недостатки параллельного программирования.

Преимущества:

- возможность выполнения нескольких задач в один момент времени;

- более высокая скорость работы приложений;

- простая и удобная структура программы.

Недостатки:

- трудности для новичков, связанные с иным типом мышления при программировании;

- неопределенность приоритета задач;

- трудности взаимосвязи между задачами;

- взаимоблокировка задач;

- проблемы при отказе оборудования;

- необходимость подбора оптимального количества процессоров.

### Понятие потокобезопасности. Причины, проблематика, способы обеспечения.

Потокобезопасность работает путем создания копии локального хранилища в каждом потоке, чтобы данные не сталкивались с другим потоком.

**Общие небезопасные ситуации потока**

* Функции, не защищающие общие переменные
* Функция, состояние которой изменяется вместе с состоянием
* Функции, возвращающие указатели на статические переменные
* Функции, вызывающие нереентерабельные функции

**Распространенные потокобезопасные ситуации**

* У каждого потока есть разрешение только на чтение для общих глобальных переменных потока или статических переменных, но нет разрешения на запись. В общем, эти потоки безопасны.
* Классы или интерфейсы - это атомарные операции для потоков
* Переключение между несколькими потоками не вызовет неоднозначности в результате выполнения интерфейса.

# **Как обеспечить безопасность потоков**

**Взаимоисключающий**: Убедитесь, что только один поток выполнения получает доступ к критическим ресурсам одновременно, то есть в любое время взаимное исключение гарантирует, что один и только один (поток) поток выполнения входит в критическую область, обращается к критическому ресурсу и реализует защиту критический ресурс.

**Синхронизировать**: Убедитесь, что программа имеет разумный доступ к критическим ресурсам (то есть после того, как один поток выполнения завершает логику, ресурсы процессора должны быть освобождены вовремя, чтобы другие потоки выполнения по-прежнему оставались их логикой задачи).

### Алгоритм выполнения многопоточной программы. Блокировка потоков.

- потоки управления (threads) образуются и работают в рамках одного процесса.

- в однопоточной программе используется только 1 главный поток (некоторая “нить” последовательно исполняемых операторов, которая начинается при запуске программы, является непрерывной и обрывается только по завершении программы)

- при использовании многопоточности, «нить» (главный поток) в некоторой точке программы делится на две, три и т.д. нити, которые в дальнейшем так же могут порождать другие потоки от себя. Но при этом **важно знать**, что один из потоков **всегда остается главным**, и его завершение означает завершение всей программы!

- в каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Несмотря на внешнюю простоту такого механизма, деятельность потоков при этом должна быть полностью согласованна. Одновременная попытка изменения двумя и более потоками объекта скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

- Классическим средством согласования потоков являются **семафоры**. Самый простой – **lock(замок)**  или **mutex(взаимоисключения)**. Если в некоторой части программы нам необходимо сохранить целостность данных согласовав потоки, то логично использовать замки. Принцип их работы можно описать как эстафету с палочкой. Пока данными пользуется один поток, он забирает «эстафетную палочку» , в то время как следующий поток ожидает передачи «палочки» от первого потока, чтобы потом начать использовать те же данные.

### Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.

,Одновременная попытка изменения двумя и более потоками объекта в многопоточной программе скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы.

Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков Lock, RLock и класс Semaphore ) и другие механизмы взаимодействия между потоками (классы Event и Condition ), класс Timer для запуска функции по прошествии некоторого времени. Модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков.

**Замки** Простейший замок может быть реализован на основе класса Lock модуля threading. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса Lock имеет следующие 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 (то есть 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). Следует иметь в виду, что даже если состояние замка только что проверено, это не означает, что он сохранит это состояние до следующей команды.

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. Например, если для корректной работы программы требуется добавление определенного элемента сразу в несколько списков или словарей, такие операции в многопоточном приложении следует обставить замками.

**Семафоры** являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода acquire() и увеличивающийся при каждом вызове release(). Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится.

Конструктор класса threading.Semaphore принимает в качестве (необязательного) аргумента начальное состояние счетчика (по умолчанию оно равно 1, что соответствует замку класса Lock ).

Методы acquire() и release() действуют аналогично описанным выше одноименным методам у замков.

**События.** Еще одним способом коммуникации между объектами являются события. Экземпляры класса threading.Event могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Если флаг в установленном состоянии, ожидания не происходит: поток, вызвавший метод wait() для ожидания события, просто продолжает свою работу.

**Условия.**

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса threading.Condition и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса

**Очередь.** Процесс, показанный в предыдущем примере, имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка Python есть, и он называется Queue.

Помимо исключений - Queue.Full (очередь переполнена) и Queue.Empty (очередь пуста) - модуль определяет класс Queue, заведующий собственно очередью.

Собственно, здесь можно привести аналог примера выше, но уже с использованием класса Queue.Queue:

### Работа с файловой системой в Python. Основные операции.(я не уверен, что это то что надо )

Модуль OS в python предоставляет функции для взаимодействия с операционной системой. OS, поставляется под стандартные служебные модули Python. Этот модуль предоставляет портативный способ использования функциональных возможностей, зависящих от операционной системы. Модули \* os \* и \* os.path \* включают в себя множество функций для взаимодействия с файловой системой.

**os.name** - имя операционной системы. Доступные варианты: 'posix', 'nt', 'mac', 'os2', 'ce', 'java'.

**os.environ** - словарь переменных окружения. Изменяемый (можно добавлять и удалять переменные окружения).

**os.getlogin**() - имя пользователя, вошедшего в терминал (Unix).

**os.getpid**() - текущий id процесса.

**os.uname**() - информация об ОС. возвращает объект с атрибутами: sysname - имя операционной системы, nodename - имя машины в сети (определяется реализацией), release - релиз, version - версия, machine - идентификатор машины.

**os.access**(path, mode, \*, dir\_fd=None, effective\_ids=False, follow\_symlinks=True) - проверка доступа к объекту у текущего пользователя. Флаги: **os.F\_OK** - объект существует, **os.R\_OK** - доступен на чтение, **os.W\_OK** - доступен на запись, **os.X\_OK** - доступен на исполнение.

**os.chdir**(path) - смена текущей директории.

**os.chmod**(path, mode, \*, dir\_fd=None, follow\_symlinks=True) - смена прав доступа к объекту (mode - восьмеричное число).

**os.chown**(path, uid, gid, \*, dir\_fd=None, follow\_symlinks=True) - меняет id владельца и группы (Unix).

**os.getcwd**() - текущая рабочая директория.

**os.link**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None, follow\_symlinks=True) - создаёт жёсткую ссылку.

**os.listdir**(path=".") - список файлов и директорий в папке.

**os.mkdir**(path, mode=0o777, \*, dir\_fd=None) - создаёт директорию. OSError, если директория существует.

**os.makedirs**(path, mode=0o777, exist\_ok=False) - создаёт директорию, создавая при этом промежуточные директории.

**os.remove**(path, \*, dir\_fd=None) - удаляет путь к файлу.

**os.rename**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает файл или директорию из src в dst.

**os.renames**(old, new) - переименовывает old в new, создавая промежуточные директории.

**os.replace**(src, dst, \*, src\_dir\_fd=None, dst\_dir\_fd=None) - переименовывает из src в dst с принудительной заменой.

**os.rmdir**(path, \*, dir\_fd=None) - удаляет пустую директорию.

**os.removedirs(path)** - удаляет директорию, затем пытается удалить родительские директории, и удаляет их рекурсивно, пока они пусты.

**os.symlink**(source, link\_name, target\_is\_directory=False, \*, dir\_fd=None) - создаёт символическую ссылку на объект.

**os.sync**() - записывает все данные на диск (Unix).

**os.truncate**(path, length) - обрезает файл до длины length.

**os.utime**(path, times=None, \*, ns=None, dir\_fd=None, follow\_symlinks=True) - модификация времени последнего доступа и изменения файла. Либо times - кортеж (время доступа в секундах, время изменения в секундах), либо ns - кортеж (время доступа в наносекундах, время изменения в наносекундах).

**os.walk**(top, topdown=True, onerror=None, followlinks=False) - генерация имён файлов в дереве каталогов, сверху вниз (если topdown равен True), либо снизу вверх (если False). Для каждого каталога функция walk возвращает кортеж (путь к каталогу, список каталогов, список файлов).

**os.system**(command) - исполняет системную команду, возвращает код её завершения (в случае успеха 0).

**os.urandom**(n) - n случайных байт. Возможно использование этой функции в криптографических целях.

### Понятие веб-технологий. Основные характеристики, история, назначение.

**Веб-технологии** — это коммуникационные, информационные и иные технологии и сервисы, основываясь на которые осуществляется деятельность в Интернете или с помощью него. В первую очередь, это, конечно, сайты, а также: чаты, почта, Интернет-магазины, форумы и т.д.

Веб–технологии — это логическая составляющая Интернет-технологий, которые включают в себя:

**1) Интернет-сервисы**

● WWW — Всемирная паутина

**2) Работа в Интернет**

● Браузеры.

● Поисковые системы.

● Просмотр страниц в браузере.

**3) Информационные ресурсы Интернет**

● Веб-страницы, интернет-магазины, интернет-порталы.

● URL и протоколы передачи данных, адресация.

● Создание сайтов.

● Языки Веб-программирования.

**Основные характеристики**

· Работает WWW по принципу клиент- серверы. То есть клиент делает запрос на серверы, серверы выдают клиенту гипермедийный текст (тексты, фото, видео, звук).

· Основные составляющие при работе с WWW: HTML (специальный язык разметки гипертекста), URL (адресная ссылка на ресурсы и их части: протокол, хост, порт, путь до ресурса, запрос), HTTP (протокол передачи гипертекста).

**Назначение**

Позволяет пользователю разместить, найти, обрабатывать практически любую информацию в сети Интернет

### Программное обеспечение, используемое для веб-технологий. Виды, назначение, примеры.

Создание web-сайтов и других web-технологий, а также их поддержка и развитие осуществляется с помощью специализированного ПО.

**Серверные ОС**

Назначение серверной операционной системы: под управлением этих ОС выполняются приложения, обслуживающие всех пользователей корпоративной сети, а нередко и внешних пользователей. К таким приложениям относятся современные системы управления базами данных, средства управления сетями и анализа событий в сети, службы каталогов, средства обмена сообщениями и групповой работы, Web-серверы, почтовые серверы, серверы приложений разнообразного назначения.

*Примеры серверных ОС*: Solaris, Linux, MS Windows Server, HP UX.

**Web-браузер**

Должно быть ПО, которое сможет запросить информацию, получить ее, обработать и отобразить на экран пользователя. Именно этим и занимается браузер.

**Web-сервер**

Чтобы опубликовать веб-сайт, нужен либо статический, либо динамический веб-сервер.

***Как он работает***

Простыми словами, когда браузеру нужен файл, размещенный на веб-сервере, браузер запрашивает его через HTTP. Когда запрос достигает нужного веб-сервера (железо), сервер HTTP (ПО) передает запрашиваемый документ обратно, также через HTTP.

1. При получении запроса, HTTP сервер сначала проверяет существует ли ресурс по данному URL.
2. Если это так, веб-сервер отправляет содержимое файла обратно в браузер. Если нет, сервер приложений создает необходимый ресурс.
3. Если это невозможно, веб-сервер возвращает сообщение об ошибке в браузер, чаще всего «404 Not Found». (Эта ошибка настолько распространена, что многие веб-дизайнеры тратят большое количество времени на разработку 404 страниц об ошибках.)

***Примеры:*** Apache, Nginx, Microsoft-IIS

Apache и Nginx — 2 самых широко распространенных веб-сервера с открытым исходным кодом в мире. Вместе они обслуживают более 50% трафика во всем интернете. Оба решения способны работать с разнообразными рабочими нагрузками и взаимодействовать с другими приложениями для реализации полного веб-стека.

***Преимущество Apache:***

Администраторы часто выбирают Apache из-за его гибкости, мощности и широкой распространенности. Он может быть расширен с помощью системы динамически загружаемых модулей и исполнять программы на большом количестве интерпретируемых языков программирования без использования внешнего программного обеспечения.

***Преимущество Nginx:***

Администраторы часто выбирают Nginx из-за его эффективного потребления ресурсов и отзывчивости под нагрузкой, а также из-за возможности использовать его и как веб-сервер, и как прокси.

**Серверы приложений**

Предназначены выполнять обширный набор специализированных функций — от обеспечения гибких средств электронных коммуникаций до управления реляционными базами данных.ё

· *Пример: WebSphere Application Server, Tomcat*

**Tomcat** позволяет запускать [веб-приложения](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5) и содержит ряд программ для самоконфигурирования.

**Tomcat** используется в качестве самостоятельного [веб-сервера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), в качестве сервера контента в сочетании с веб-сервером [Apache HTTP Server](https://ru.wikipedia.org/wiki/Apache_HTTP_Server), а также в качестве контейнера сервлетов в [серверах приложений](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) [JBoss](https://ru.wikipedia.org/wiki/JBoss) и [GlassFish](https://ru.wikipedia.org/wiki/GlassFish).

### Понятие URL: назначение, применение, состав.

**URL** – это адрес документа или файла. Аббревиатура URL составлена из начальных букв исходного сочетания Uniform Resource Locator, что в переводе с английского языка означает понятие Единый указатель ресурсов.

Другими словами, он определяет местонахождение ресурса либо документа в сети, его сетевой локатор, ссылка на него.

Каждая интернет-страница расположена на собственном уникальном адресе, который пользователь может видеть в специальном поле – адресной строке браузера – он и указывает, где хранятся эти файлы в сети. И урл есть не что иное, как общепринятая стандартная форма записи этого местонахождения во Всемирной паутине.

Чтобы осуществлять взаимодействие в глобальной сети, обмениваться данными, иметь доступ к искомому URL веб-ресурса, используют протоколы – так называется свод правил и очередности действий, который позволяет устанавливать соединение устройств в Интернете.

Практически мгновенная предоставление информации посредством Всемирной паутины обеспечивается благодаря поддержке протоколами TCP/IP, ATM, MPLS и прочими.

Самым распространенным среди них является набор протоколов передачи данных TCP/IP.

В их числе наибольшей популярностью в сети пользуются:

● HTTP — это протокол передачи гипертекста. Он применяется при обмене данными с серверов – подключенному к Интернету компьютеру пользователя по его запросу;

● FTP — протокол, назначение которого — передача документов с сервера на персональное устройство пользователя. FTP позволяет обмениваться документами любым компьютерам сети.

Каждый URL **состоит** из определенной последовательности символов. Это не хаотичный набор знаков, а определенная структура, которая составляется по четкой схеме.

По задумке создателей, сетевой локатор необходимо максимально упростить, чтобы тот включал в себя небольшое число символов.

Его составляющие в ниже представленной последовательности могут быть следующими:

способ обмена данными с ресурсом или тип сетевого протокола;

логин, то есть то имя учетной записи пользователя для обращения к ресурсу;

пароль для входа в учетную запись;

хост или целиком вписанное доменное имя;

порт для подключения в пределах обозначенного хоста;

URL-путь – содержит конкретизирующую информацию о заданных характеристиках места расположения ресурса;

строка запроса, содержащая передаваемые на сервер параметры;

«якорь», в функцию которого входит возможность ссылаться на определенную часть документа, страницы.

Такая последовательность указывает компьютерной системе, где конкретно хранится необходимый пользователю ресурс и путь к нему.

Если на заре создания URL служил для определения пути к определенным файлам, то в настоящее время посредством ссылок обозначается расположение любого типа информации:

● интернет-ресурсов;

● веб-страниц;

● изображений;

● видео.

Например, чтобы просмотреть видео на сайте, компьютер обращается к точному адресу места хранения документа. Если файл удалятся правообладателем или URL изменяют, просмотреть данное видео пользователь не сможет.

Применение идентификатора URL дает возможность создавать комфортные, понятные для восприятия пользователя конструкции символов. Благодаря чему в этом случае исчезает необходимость использовать программные коды.

Вместе с тем помимо этих достоинств существуют некоторые ограничения. В URL

### Понятие веб-сервера. Цели, принцип работы. **Веб-сервер с точки зрения ОС**

### Веб сервер - это обычный сетевой демон, прослушивающим входящие соединения на порту 80. Он принимает запросы на выдачу ресурсов в соответствии с протоколом http, а затем высылает этим документы пользователю.

**Ресурсы веб-сервера**

Статические и динамические ресурсы

HTML-документы, картинки, стили, файлы JS, другие типы.

При обращении к программам определенного вида они генерируют страницы как свой вывод.

Ресурсы размещаются в определенном каталоге на компьютере сервера.

**Функции веб-сервера**

Общение по протоколу http

Доступ к части файловой системы компьютера-сервера.

Запуск сторонних программ.

Обеспечение безопасности.

Логирование обращений пользователя.

Встроенные интерпретаторы языков программирования

Распределение нагрузки

### 

### Протокол HTTP. Принцип работы, назначение, основные понятия.

HTTP (HyperText Transfer Protocol) – это протокол прикладного уровня для передачи гипермедиа документов, таких как HTML. Он был разработан для связи между веб-браузерами и веб-серверами, но также может использоваться для других целей. HTTP следует классической модели клиент-сервер: клиент открывает соединение, чтобы сделать запрос, а затем ждет, пока не получит ответ.



Типичные задачи, которые решает HTTP протокол: протокол HTTP осуществляет доступ к веб-ресурсам и обмен данными между пользовательскими приложениями.

Передача данных по HTTP протоколу осуществляется через TCP/IP соединение. Машина, которая выступает в роли сервера использует восьмидесятый TCP порт или порт 8080.

Функции, управляемые с помощью HTTP:

1) Кеширование

2) Аутентификация

3) Proxy

4) Ослабление ограничения источника.

5) Сеансы

Структура протокола.

Структура протокола определяет, что каждое HTTP-сообщение состоит из трёх частей, которые передаются в следующем порядке:

1. Запрос — определяет тип сообщения;

2. Поля запроса — характеризуют тело сообщения, параметры передачи и прочие сведения;

3. Тело сообщения (англ. Message Body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

Методы протокола

Метод HTTP (англ. HTTP Method) — последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Названия метода чувствительны к регистру. HTTP метод , как правило , имеет вид - GET, POST , OPTIONS или HEAD что определяет операцию клиент хочет выполнить. Как правило, клиент хочет получить ресурс (используя GET) или опубликовать значение HTML-формы (используя POST), хотя в других случаях может потребоваться больше операций.

Примеры методов протокола:

1) GET

2) POST (Запостить файл – создать файл)

3) PUT

4) DELETE (удалить) и др.

Код состояния информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Набор кодов состояния является стандартом, и все они описаны в соответствующих документах RFC.

Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие - порядковый номер состояния. За кодом ответа обычно следует краткое описание на английском языке.

### Настройка веб-сервера. Основные конфигурационные файлы, понятия.

### **1. Установка**

### apt install apache2

### После завершения установки откроем браузер и перейдем по ссылке “http://[ip\_адрес\_сервера]”

### **2. Создание тестовой страницы**

### По умолчанию корневым каталогом для размещения сайта является директория “/var/www/html”, именно там находится страница приветствия. Создадим отдельную

### директорию “/var/www/sites” для размещения виртуальных хостов и вложенную папку “/var/www/sites/site1” с индексной страницей тестового сайта.

### cd /var/www/

### mkdir -p sites/site1

### echo "<H1>Welcome</H1>" > sites/site1/index.html

### В результате файл “/var/www/sites/site1/index.html” будет содержать одну html-строку:

### <H1>Welcome</H1>

### **3. Конфигурация Apache-сервера**

### Конфигурационные файлы сайтов находятся в каталоге “/etc/apache2/sites-available/”. Создадим конфигурационный файл для нового виртуального хоста взяв за основу конфигурацию по умолчанию из файла “000-default.conf”

### Откроем файл “site1.conf” и изменим параметр “DocumentRoot”. В качестве значения нужно указать путь к новому сайту, в нашем случае это “/var/www/sites/site1”

### На данном этапе нам не требуется настройка одновременной работы нескольких сайтов, поэтому отключим сайт по умолчанию и включим новый сайт. Для применения изменений перезагружаем конфигурацию сервера.

### a2dissite 000-default

### a2ensite site1

### systemctl reload apache2

### Снова переходим по ссылке “http://[ip\_адрес\_сервера]” и убеждаемся, что вместо стандартной страницы приветствия отображается наша новая страница.

### Настройка HTTP-сервера завершена.

### В зависимости от версии Apache и вашего дистрибутива, конфигурационные файлы Apache могут находиться в следующих каталогах: /etc/apache, /etc/apache2, /etc/httpd или /etc/httpd2. Основные конфигурационные файлы называются httpd.conf, httpd2.conf или apache.conf и apache2.conf.

### Основные директивы и их описание

### ServerName имя - задает имя Web-сервера, имя должно быть зарегистрированным на DNS-сервере, то есть обычно это доменное имя сервера

### ServerAdmin e-mail - задает e-mail администратора сервера

### ServerRoot каталог - определяет каталог с конфигурационными файлами сервера

### DocumentRoot каталог - Позволяет задать каталог, в котором хранятся документы Web-сервера - это корневой каталог документов.

### ErrorLog файл - задает журнал ошибок

### User пользователь

### Group группа

### 

### Виртуальные хосты. Применение, настройка.

Что такое виртуальные хосты?

Базовая часть, которая отвечает за отдельный сайт или домен называется виртуальным хостом

Эта система позволяет использовать один сервер, чтобы раздавать несколько сайтов используя один интерфейс или IP

Каждый настроенный соответствующим образом домен будет направлять пользователя к определенной директории сервера.

### **Применение, настройка**

# создание директорий

$ mkdir -p /var/www/example.com/public\_html

$ sudo mkdir -p /var/www/test.com/public\_html

# назначение прав

$ sudo chown -R $USER:$USER /var/www/example.com/public\_html

$ sudo chown -R $USER:$USER /var/www/test.com/public\_html

$ sudo chmod -R 755 /var/www

# создать соответствующие файлы сайтов

Создание файлов виртуального хоста

# /etc/apache2/sites-available/example.com.conf

<VirtualHost \*:80>

ServerAdmin admin@example.com

ServerName example.com

ServerAlias www.example.com

DocumentRoot /var/www/example.com/public\_html

ErrorLog ${APACHE\_LOG\_DIR}/error.log

CustomLog ${APACHE\_LOG\_DIR}/access.log combined

</VirtualHost>

Включение новых виртуальных хостов

sudo a2ensite example.com.conf

sudo a2ensite test.com.conf

sudo a2dissite 000-default.conf

sudo systemctl restart apache2

Настройка файла локальных хостов

$ sudo nano /etc/hosts

127.0.0.1 localhost  
127.0.1.1 guest-desktop  
111.111.111.111 example.com  
111.111.111.111 test.com

### Понятие прокси-сервера. Настройка сервера nginx.

Рroxy-сервер – это дополнительная служба в сети интернет, позволяющая пользователям выполнять косвенные запросы к различным веб-ресурсам и сетевым службам. Данный ресурс выступает в качестве посредника, причем на оперативности и качестве работы это отражается только в лучшую сторону, за счет использования отдельного оптоволокна (выделенного канала) с максимально возможной скоростью обмена данными.

**Основные и наиболее часто используемые возможности прокси-серверов:**

· Ускорение работы с интернет-ресурсами за счет кэширования (большинство контента, фото- и видеоматериала уже загружено на дополнительном промежуточном сервере).

· Защита программного обеспечения компьютера пользователя от всевозможных сетевых атак (вирусов).

· Анонимность посещения всех интернет ресурсов. При использовании proxy-серверов все запросы на веб-ресурсы будут проходить под IP адресом прокси-сервера и информация о пользователе (страна, регион, провайдер, номер компьютера и адрес) будет завуалирована.

· Открытый доступ пользователя к социальным сетям, чатам и сайтам, даже если аккаунт заблокирован системным администратором (модератором) на уровне IP-адреса. Система будет воспринимать данный запрос к доступу как от нового пользователя, так как IP адрес будет изменен.

· Доступ к информации на сайтах с ограничениями на просмотр. Некоторые сайты могут устанавливать блокировку по определенному критерию или географическому положению. Пользователю достаточно подключиться к прокси-серверу данной страны, и система проверки IP адреса не распознает подмены, так как IP адрес будет соответствовать географическому положению.

· Фильтрация рекламы (автоматическое удаление рекламного контента с просматриваемых страниц).

**Понятие Прокси-сервера**

Прокси-сервер — удаленный компьютер, который, при подключении к нему вашей машины, становится посредником для выхода абонента в интернет. Прокси передает все запросы программ пользователя в сеть, и, получив ответ, отправляет его обратно.

Каждому компьютеру, подключенному к интернет, присваивается уникальный ip-адрес, который несет информацию о стране и регионе абонента, номере его провайдера и номере компьютера в сети.

Прокси-сервер также имеет свой ip-адрес. Подключившись к прокси, вы передаете все запросы в интернет через него, при этом проверка показывает ip прокси-сервера, а вы остаетесь анонимным.

**Настройка nginx**

Nginx доступен в стандартных репозиториях ubuntu, а установить его мы сможем при помощи пакетов APT. Для начала обновим список пакетов apt, а потом установим nginx:

sudo apt update && sudo apt install nginx

Далее необходимо настроить файрвол. При установки Nginx регистрируется в сервисе файрвола ufw. Поэтому настройка доступа осуществляется достаточно просто. Для вывода настроек доступа введем команду

sudo ufw app list

В результате будет выведен список профилей приложений:

Nginx Full

Nginx HTTP

Nginx HTTPS

Выбираем нужный, и пишем команду:

sudo ufw allow 'Nginx HTTP'

Далее ubuntu запустит nginx автоматически. Убедимся, введя команду

systemctl status nginx

Таким образом, сервер запущен и работает. Тем не менее, лучше проверить его работоспособность путем запроса веб-страницы.

Для остановки, старта, и перезапуска сервера необходимо ввести следующие команды:

sudo systemctl stop nginx

sudo systemctl start nginx

sudo systemctl restart nginx

### Основные принципы криптографии. Шифры. Исторические шифры.

Шифрование - преобразовательный процесс: исходный текст, который носит также название открытого текста, заменяется шифрованным текстом.

Дешифрование - обратный шифрованию процесс. На основе ключа шифрованный текст преобразуется в исходный.

Криптографические методы защиты информации - это специальные методы шифрования, кодирования или иного преобразования информации, в результате которого ее содержание становится недоступным без предъявления ключа криптограммы и обратного преобразования.

**Задачи**:

· Обеспечение секретности

· Аутентификация сторон

· Обеспечение целостности

**Шифры**:

Шифрование – процесс криптографического преобразования текста на основе определенного параметра (ключа) и алгоритма. Расшифрование – криптографическое преобразование шифрованного текста в исходный. Шифр – совокупность обратимых преобразований исходных данных в скрытый текст.

**Виды**:

Симметричное шифрование (шифрование с закрытым ключом)

Есть ключ шифрования. С его помощью данные шифруются по какому-то алгоритму. Тот, кто обладает ключом и знает алгоритм, может расшифровать сообщение.

Асимметричное шифрование (шифрование с открытым ключом)

Асимметричное шифрование — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом.

Электронные подписи(MAC, HMAC)

Электронная подпись — это асимметричное шифрование наоборот: вы зашифровываете закрытым ключом, а расшифровать может кто угодно с помощью открытого ключа, который доступен всем.

Хэширование(MD5, SHA1, SHA256)

Криптографические хэши используются везде, от хранения паролей до систем проверки файлов. Основная идея состоит в том, чтобы использовать детерминированный алгоритм (алгоритмический процесс, который выдает уникальный и предопределенный результат для задачи входных данных), который принимает один вход и создает строку фиксированной длины каждый раз.

### Симметричное шифрование. Примеры алгоритмов, общая схема, виды.

**Симметричное шифрование** — это способ шифрования данных, при котором один и тот же ключ используется и для кодирования, и для восстановления информации.

В целом симметричным считается любой шифр, использующий один и тот же секретный ключ для шифрования и расшифровки. Например, если алгоритм предполагает замену букв числами, то и у отправителя сообщения, и у его получателя должна быть одна и та же таблица соответствия букв и чисел: первый с ее помощью шифрует сообщения, а второй — расшифровывает

Примеры

AES (блочное)

DES (блочное ?)

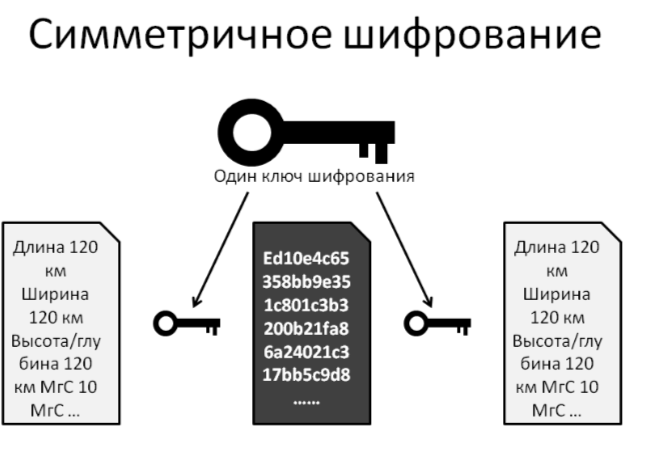
3DES (симметричный [блочный шифр](https://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%88%D0%B8%D1%84%D1%80))

Blowfish (блочное)

RC4 (потоковое)

Общая схема симметричного шифрования

Сообщение → Шифрованное сообщение → сообщение



Виды симметричного шифрования

В зависимости от принципа работы алгоритмы симметричного шифрования делятся на два типа:

*блочные;*

*потоковые*.

*Блочные* алгоритмы шифруют данные блоками фиксированной длины (64, 128 или другое количество бит в зависимости от алгоритма). Если все сообщение или его финальная часть меньше размера блока, система дополняет его предусмотренными алгоритмом символами, которые так и называются *дополнением*.

К актуальным блочным алгоритмам относятся:

* [AES](https://encyclopedia.kaspersky.ru/glossary/aes/)
* ГОСТ 28147-89
* RC5
* Blowfish
* Twofish

Потоковое шифрование данных предполагает обработку каждого бита информации с использованием *гаммирования*, то есть изменения этого бита с помощью соответствующего ему бита псевдослучайной секретной последовательности чисел, которая формируется на основе ключа и имеет ту же длину, что и шифруемое сообщение. Как правило, биты исходных данных сравниваются с битами секретной последовательности с помощью логической операции XOR (исключающее ИЛИ, на выходе дающее 0, если значения битов совпадают, и 1, если они различаются).

Потоковое шифрование в настоящее время используют следующие алгоритмы:

* [RC4](https://encyclopedia.kaspersky.ru/glossary/rc4/)
* Salsa20
* HC-256
* WAKE

### Асимметричное шифрование. Примеры алгоритмов, общая схема, преимущества и недостатки.

Асимметричное шифрование

**Асимметричное шифрование** — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом. Открытый и закрытый ключи — это очень большие числа, связанные друг с другом определенной функцией, но так, что, зная одно, крайне сложно вычислить второе.

Асимметричное шифрование используется для защиты информации при ее передаче, также на его принципах построена работа электронных подписей.

Примеры алгоритмов

**RSA**

DSA

ECDSA

Подробнее о примерах, что это ваще:

Наиболее распространенные алгоритмы асимметричного шифрования:

* [RSA](https://encyclopedia.kaspersky.ru/glossary/rsa/) (аббревиатура от Rivest, Shamir и Adelman, фамилий создателей алгоритма) — алгоритм, в основе которого лежит вычислительная сложность факторизации (разложения на множители) больших чисел. Применяется в защищенных протоколах SSL и TLS, стандартах шифрования, например в PGP и S/MIME, и так далее. Используется и для шифрования данных, и для создания цифровых подписей.
* DSA (Digital Signature Algorithm, «алгоритм цифровой подписи») — алгоритм, основанный на сложности вычисления дискретных логарифмов. Используется для генерации цифровых подписей. Является частью стандарта DSS (Digital Signature Standard, «стандарт цифровой подписи»).
* Схема Эль-Гамаля — алгоритм, основанный на сложности вычисления дискретных логарифмов. Лежит в основе DSA и устаревшего российского стандарта ГОСТ 34.10–94. Применяется как для шифрования, так и для создания цифровых подписей.
* ECDSA (Elliptic Curve Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретного логарифма в группе точек эллиптической кривой. Применяется для генерации цифровых подписей, в частности для подтверждения транзакций в криптовалюте Ripple.

Общая схема



Преимущества и Недостатки

Преимущества

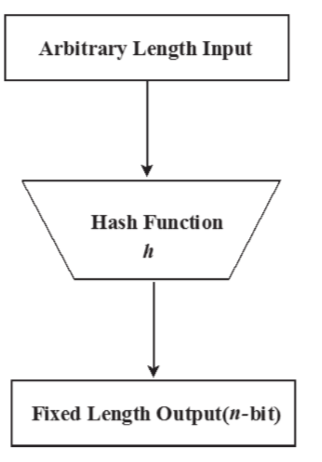
отсутствие проблем с передачей ключа

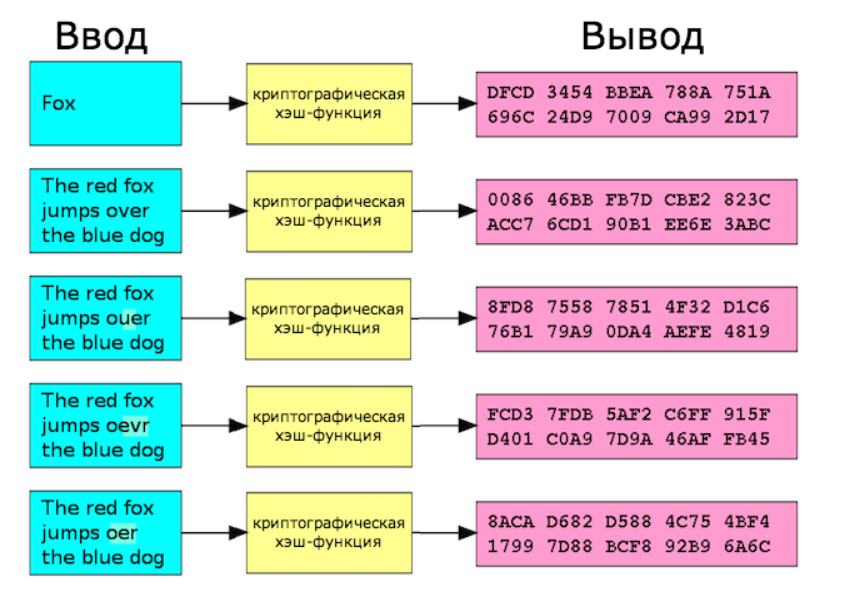
упрощенная процедура хранения пароля (отсутствие необходимости передавать кому-то закрытый ключ).

Недостатки

сложности в понимании работы системы; процедура шифрования и дешифрования требует от компьютера, на котором она проводится, значительных ресурсов.

### Алгоритмы хэширования. Примеры, назначение.





## **Что такое хеш (хэш, hash)?**

Хеш или хэш — это криптографическая функция хеширования (function), которую обычно называют просто хэшем. Хеш-функция представляет собой математический алгоритм, который может **преобразовать произвольный массив данных в строку фиксированной длины**, состоящую из цифр и букв.

### **Протокол TLS/SSL. Общая схема взаимодействия, назначение.**

Протокол TLS (transport layer security) основан на протоколе SSL (Secure Sockets Layer), изначально разработанном в Netscape для повышения безопасности электронной коммерции в Интернете. Протокол SSL был реализован на application-уровне, непосредственно над TCP (Transmission Control Protocol), что позволяет более высокоуровневым протоколам (таким как HTTP или протоколу электронной почты) работать без изменений. Если SSL сконфигурирован корректно, то сторонний наблюдатель может узнать лишь параметры соединения (например, тип используемого шифрования), а также частоту пересылки и примерное количество данных, но не может читать и изменять их.

Протокол TLS преследует три цели:

обеспечение безопасности  
 соединения

идентификация сторон

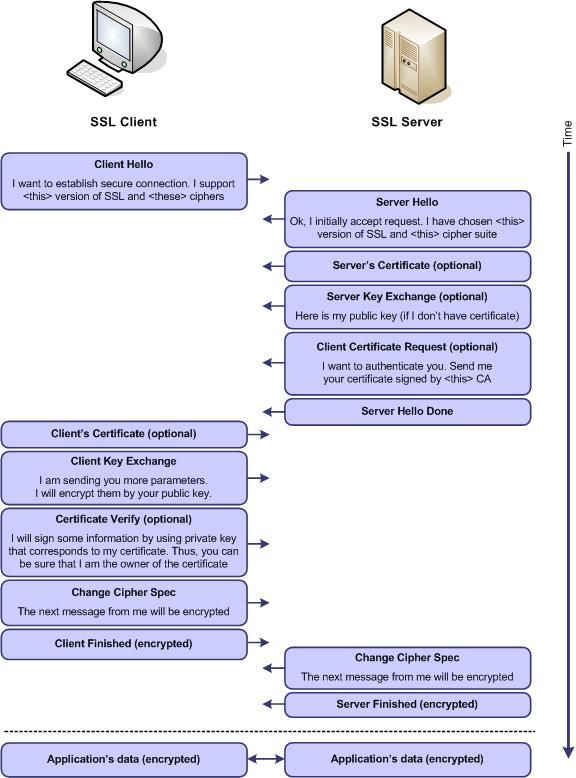
обеспечение надежной передачи данных.

Для работы TLS/SSL использует комбинацию открытого сертификата и закрытого ключа. Закрытый ключ хранится на сервере и не разглашается. SSL-сертификат используется открыто и доступен всем пользователям, запрашивающим контент

схема работы

Протокол реализуется на базе стека TCP/IP

Устанавливает алгоритмы шифрования и ключи на обеих сторонах и прокидывает шифрованный туннель, по которому могут передаваться другие протоколы (например HTTP)



назначение

**Протоколы SSL/TLS** предназначены для исключения подмены идентификатора на клиентской или серверной стороне, раскрытия или искажения данных. Для этих целей используется надежный метод аутентификации, применяются шифрование канала связи и коды целостности сообщений.

### **Понятие SSL-сертификата. Назначение. Самоподписанные сертификаты. Центры сертификации.**

**SSL-сертификат (*Secure Sockets Layer* — уровень защищённых** [**сокетов**](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81))**)** - это виртуальный документ, подтверждающий подлинность сайта и позволяющий узнать его владельца и название организации. SSL-сертификат необходим для установки безопасного соединения HTTPS, при установке которого сервер предъявляет ключ - это и есть SSL-сертификат. В общем, сертификат SSL - доказательство подлинности и безопасности сайта. После того, как браузер подтвердил подлинность сертификата сервера, они обмениваются ключами шифрования и продолжают общение в безопасном режиме. Ключи обмена шифруются ассиметричным способом, а сами сообщения - симметрично.

(======== доп инфа - можно расписать, но не обязательно) Протокол SSL предоставляет «безопасный канал», который имеет три основных свойства:

1. Канал является частным. Шифрование используется для всех сообщений после простого диалога, который служит для определения секретного ключа.
2. Канал аутентифицирован. Серверная сторона диалога всегда аутентифицируется, а клиентская делает это опционально.
3. Канал надёжен. Транспортировка сообщений включает в себя проверку целостности.

Преимуществом SSL является то, что он независим от прикладного протокола. Протоколы приложений ([HTTP](https://ru.wikipedia.org/wiki/HTTP), [FTP](https://ru.wikipedia.org/wiki/FTP), [TELNET](https://ru.wikipedia.org/wiki/TELNET) и т. д.) могут работать поверх протокола SSL совершенно прозрачно, то есть SSL может согласовывать алгоритм шифрования и ключ сессии, а также аутентифицировать сервер до того, как приложение примет или передаст первый байт сообщения.

На данный момент непосредственно протокол SSL является устаревшим и его использование сокращается в пользу TLS (т. к. он основан на SSL и по факту является его усовершенствованной версией, его часто продолжают называть по имени “предка”). (==================)

**Самоподписанный сертификат** — сертификат, созданный самим пользователем — в этом случае издатель сертификата совпадает с владельцем сертификата. По сравнению с заверенным сертификатом, у него есть ряд плюсов и минусов. Плюсы: бесплатность, меньшие времязатраты (в случае наличия опыта создания сертификатов), возможность создавать неограниченное количество сертификатов без обращения за услугами. Главный минус - работать с такими сертификатами можно только внутри своей организации, т. к. их необходимо вручную добавлять в браузер, чтобы сайт идентифицировался как безопасный.

Другой способ получить SSL-сертификат - обратиться в центр сертификации. **Центр сертификации SSL** - авторитетная международная компания, которой доверяет большинство браузеров, например, Comodo, GeoTrust и Thawte. Компании дорожат своей репутацией и перед установкой электронной подписи на сертификате проводят ряд проверок домена и организации. За услуги сертификации необходимо заплатить, но сумма, как правило, не очень большая, а рейтинг сайта в поисковике притом поднимается довольно сильно.

### **FTP-сервер. Назначение, общая схема работы. Обеспечение безопасности.**

**FTP-сервер** - сервер, работающий по протоколу FTP (File Transfer Protocol) - протоколу передачи файлов по сети. В общем смысле является сервером (компьютером), который хранит и позволяет обмениваться файлами участников локальной или глобальной сети. Из ныне популярных протоколов FTP является самым старым, и, хоть он претерпел существенные изменения со времени своего создания в 1971 году, суть осталась прежней.

Особенности FTP-сервера:

* отдельный канал для каждого соединения;
* бинарный и текстовый режим передачи информации;
* необходимость аутентификации пользователя;
* способность определять тип файла.

Главное **назначение FTP-сервера** - удалённая передача и управление файлами. Он может быть использован как для личных (хранение общих с друзьями файлов на общем сервере), так и для корпоративных нужд (обмен файлами между работниками одной компании в рамках одной сети).

**Схема работы FTP-протокола.** Проводится аутентификация, в зависимости от логина пользователя ему предоставляется доступ к определённым папкам (притом даже если кажется, что аутентификации не было, она была - в таком случае автоматически подставляется рандомный e-mail и пароль anonymous). Т. к. FTP-сервер работает в множественном режиме, один канал берёт на себя роль управляющего - принимает команды и возвращает ответы (как правило, 21-й порт), а остальные каналы передают непосредственно файлы (20-й порт).

Обмен информацией может происходить в нескольких режимах: поточном (вся информация идёт сплошным потоком и никак не обрабатывается), блочном (информация блоками “заголовок - объём файла - сам файл”) и режиме сжатия (отправляется файл, сжатый по тому или иному алгоритму).

**Безопасность.** Протокол FTP довольно надёжен, когда дело касается целостности данных, но сам по себе незащищённый - это значит, что все данные, включая логины и пароли, передаются в незашифрованном виде. Если кто-то подключится к вашей локальной сети, то вполне может скачать оттуда файлы. Также FTP-сервер можно взломать подбором паролей. Для обеспечения безопасности FTP-серверов используются более современные - шифрованные - вариации протокола FTP - FTPS/SFTP. Также как вариант можно использовать FTP через VPN.

### **Настройка FTP-сервера в Linux. Основные понятия, конфигурационные файлы.**

В современные версии линукса входит полнофункциональный FTP-сервер vsftpd. Устанавливаем его (apt-get install vsftpd - в режиме root). Перед началом работы с сервером стоит добавить его в автозагрузку, а также добавить строку echo "/usr/sbin/nologin" >> /etc/shell в shell, чтобы пользователи могли подключаться к серверу без предоставления им каждый раз по-новой доступа в shell.

Настройку любого FTP-сервера можно выполнить с помощью конфигурационных файлов. Как правило, конфигурационный файл - это обычный текстовый файл, который тем или иным образом интерпретирует операционная система. В случае vsftpd конфигурационный файл всего один и он находится по этому пути: /etc/vsftpd.conf.

В этом конфигурационном файле можно настроить множество параметров, например запретить доступ анонимным пользователям, запретить скачивание файлов с сервера, запустить сервер в режиме демона (отдельный, несовместимый с предыдущим параметр - запуск в режиме демона но с прослушиванием также IPv6). Соответствующий пример:

**anonymous\_enable=NO**

**download\_enable=NO**

**listen\_ipv6=YES**

Все эти команды не обязательно помнить наизусть, достаточно посмотреть конфигурационный файл.

### **Email-сервер. Назначение, общая схема работы. Обеспечение безопасности.**

**Email-сервер** - сервер передачи электронных сообщений от одного компьютера к другому. Может обслуживать как локальную, так и глобальную сеть. Сервер электронной почты также часто называется агентом пересылки сообщений - Mail Transport Agent (MTA). Пользователи имеют дело с клиентом электронной почты, почтовым агентом - Mail User Agent (MUA). Клиент и сервер взаимодействуют по правилам, определяемым протоколом - Mail Delivery Agent (MDA), например, SMTP.

**Назначение** почтового сервера - переброс письма от отправителя к получателю.

**Схема работы.** Когда пользователь через клиент отправляет сообщение, клиент и сервер (MUA и MTA соответственно) взаимодействуют по протоколу SMTP. Почтовый сервер отправителя пытается отправить сообщение либо на сервер получателя, либо на промежуточный сервер - релей, посредством агента доставки сообщений MDA (может быть частью упомянутых здесь и далее протоколов или отдельным ПО). Клиент-получатель принимает сообщение от своего сервера по, как правило, уже другим протоколам - POP/IMAP.

**Обеспечение безопасности email-сервера** происходит благодаря шифрованию писем. Для шифрования широко применяются два стандарта: S/MIME (использует инфраструктуру открытых ключей) и OpenPGP (сертификаты по схеме доверия, группирующиеся вокруг пользователя). Они позволяют обеспечить защиту от изменения, конфиденциальность писем и неотзывную подпись. Однако они не защищают от человеческого фактора - например, фишинговых ссылок, рассылаемых по почте, заражённых файлов и так далее. Многие из этих писем блокируются почтовыми сервисами, но системы не идеальны и неопытные пользователи могут случайно попасться на уловки мошенников.

### **Настройка Email-сервера в Linux. Основные понятия, конфигурационные файлы.**

**Основные понятия.** Почтовая система состоит из трёх компонентов: почтового сервера MTA (Mail Transfer Agent), клиента MUA (Mail User Agent) и протокола передачи сообщений между серверами MDA (Mail Delievery Agent). Для организации почтового сервера необходимо установить и настроить почтовый сервер. Встроенные в линукс пакеты почтовых серверов - sendmail и postfix - основаны на SMTP.

**Настройка.** Пакет postfix более безопасен, проверим, есть ли он в системе, и если нет, то установим его (в режиме root-пользователя):

$ rpm -qa | grep postfix

$ dnf -y **install** postfix

(или, если система основана на Debian, то вот так: $ apt-**get** -y install **postfix**)

Запустим службу и добавим её в автозагрузку:

При установке можно будет выбрать один из четырёх режимов в зависимости от назначения сервера.

Далее можно начать настраивать сервер через конфигурационные файлы, которые лежат в папке /etc/postfix/. Самый главный из них - /etc/postfix/main.cf, в нём лежит множество параметров. Например, myhostname - имя хоста в Интернете, на адрес которого будет приходить почта, mydomain - имя домена, myorigin - доменное имя и т. д. Настраивается таким образом:

**myhostname** = mail.example.com

### Основные принципы мониторинга сетевых служб. Мониторинг четырех золотых сигналов.

Мониторинг — это создание, сбор, объединение и использование метрик, которые позволяют получить представление о состоянии системы.

Для мониторинга системы нам нужна информация о ее программных и аппаратных компонентах. Такую информацию можно получить через метрики, собранные с помощью специальной программы или инструментирования кода.

Инструментирование — это изменение кода таким образом, чтобы можно было измерить его производительность. Мы добавляем код, который не влияет на функции самого продукта, а просто вычисляет и предоставляет метрики. Допустим, мы хотим измерить задержку запроса. Добавляем код, который посчитает, сколько времени нужно сервису, чтобы обработать полученный запрос.

### Мониторинг четырех золотых сигналов.

### Задержка - Сколько времени занимает обработка запроса? Определите ориентир для задержек, типичных для успешных запросов, и сравните его с задержками для неуспешных запросов. Отслеживание задержек, вызванных ошибками, позволяет решить любые вопросы, связанные со скоростью выявления инцидента и реакции на него.

### Трафик - Этот сигнал не требует особых пояснений. Какое влияние на систему оказывает количество пользователей или число транзакций, проходящих через сервис? В зависимости от функциональности сервиса измерение трафика может существенно отличаться от компании к компании. Отслеживая взаимодействие с реальными пользователями и трафик, можно лучше понять, как конечные пользователи воспринимают сервис, и получить представление о том, как системы ведут себя в условиях стресса.

### Ошибки - Конечно, каждая команда должна следить за ошибками. Независимо от того, вызваны ли ошибки заданной вручную логикой или автономны (вроде неудавшегося HTTP-запроса), SRE-команды должны отслеживать их. Многие SRE-команды используют специальное [ПО для управления инцидентами](https://victorops.com/ebooks/incident-management-buyers-guide/) для оповещений о критических ошибках, поиска их причин и проведения работ по устранению последствий.

### Насыщенность - Каждая команда должна следить за загруженностью своей системы. Важно задать метрику для насыщенности, которая бы означала, что сервис достиг максимума своих возможностей. Большинство сервисов начинают терять производительность еще до того, как загрузка достигнет 100%, поэтому понимание функциональности вашей собственной системы важно для определения ориентира насыщенности, который имеет смысл.

Установив правила мониторинга и оповещений для четырех золотых сигналов, вы охватите большинство ключевых инцидентов в системе.

### Прикладные интерфейсы программирования. Назначение, виды, реализация.

Программный интерфейс приложения, или интерфейс прикладного программирования (API) – это код, который позволяет двум программам взаимодействовать друг с другом. API определяет правильный способ для разработчика запрашивать службы из операционной системы (ОС) или другого приложения и предоставлять данные в разных контекстах и по нескольким каналам.

API-интерфейсы состоят из двух связанных элементов. Первый – это спецификация, описывающая, как происходит обмен информацией между программами в форме запроса на обработку и возврата необходимых данных. Второй – это программный интерфейс, написанный в соответствии с этой спецификацией и каким-то образом опубликованный для использования. Считается, что программное обеспечение, которое хочет получить доступ к функциям и возможностям API, «вызывает» его, а программное обеспечение, которое создает API, «публикует» его.

API-интерфейсы – это, по сути, набор правил. Они могут улучшить внутренние процессы разработки организации за счет стандартизации того, как разработчики пишут код приложения – использование одних и тех же правил и форматов делает код более оптимизированным и прозрачным.

Преимущества API -

Безопасный контроль и управление доступом пользователей и систем к определенным данным и функциям услуг;

Возможность позволять третьим сторонам использовать свои данные, что увеличивает узнаваемость бренда компании;

Расширять свою клиентскую базу данных и даже повышать коэффициент конверсии, согласовывая свои услуги с услугами других надежных брендов; Монетизировать свои API, чтобы они стали отдельной статьей дохода.

Типы API -

Частные API-интерфейсы или внутренние API-интерфейсы публикуются внутри компании для использования разработчиками компании для улучшения ее собственных продуктов и услуг. Частные API не доступны третьим лицам;

Общедоступные или открытые API-интерфейсы публикуются публично и могут использоваться любой третьей стороной. Для этих API нет никаких ограничений;

Партнерские API могут использоваться только определенными сторонами, с которыми компания соглашается делиться данными. Партнерские API используются в рамках деловых отношений, часто для интеграции программного обеспечения между партнерскими компаниями;

Составные API-интерфейсы объединяют несколько API-интерфейсов для решения связанных или взаимозависимых задач и часто повышают скорость и производительность по сравнению с отдельными API-интерфейсами.

Подтипы API -

Локальные API предлагают сервисы ОС или промежуточного программного обеспечения для прикладных программ. API Microsoft .NET, TAPI (Telephony API) для голосовых приложений и API доступа к базе данных являются примерами локальной формы API; Веб-API предназначены для представления широко используемых ресурсов, таких как HTML-страницы, и доступ к ним осуществляется с помощью простого протокола HTTP. Любой веб-URL активирует веб-API. Веб-API часто называют RESTful, потому что издатель интерфейсов REST не сохраняет никаких данных внутри между запросами. Таким образом, запросы от многих пользователей могут смешиваться, как если бы они были в Интернете;

Программные API основаны на технологии удаленного вызова процедур (RPC), которая заставляет удаленный программный компонент казаться локальным для остальной части программного обеспечения. API-интерфейсы сервис-ориентированной архитектуры (SOA), такие как API-интерфейсы Microsoft WS, являются программными API-интерфейсами.

Веб-API можно вызывать через любой язык программирования, но также можно получить доступ к веб-страницам, созданным в HTML или инструментах генератора приложений. Возросшая роль Интернета и облака в нашей жизни и деловой активности значительно расширила использование API-интерфейсов и использование простых инструментов программирования или даже отказа от программирования для доступа к API.

И REST, и SOAP могут вызывать облачные сервисы, подключаться к ним и взаимодействовать с ними, но REST все чаще предпочтительнее для веб-API, поскольку он использует меньшую пропускную способность и предлагает больше возможностей для языков программирования, таких как JavaScript или Python.

Тенденция рассматривать API как представление общих ресурсов изменилась. В то время как ожидается, что API-интерфейсы будут использоваться в качестве общего инструмента многими приложениями и пользователями, они считаются службами и обычно требуют более контролируемой разработки и развертывания. SOA и микросервисы являются примерами сервисных API. Сервисы – самая горячая тенденция в API, до такой степени, что возможно, что все API в будущем будут рассматриваться как представляющие сервисы.

### Развертывание приложений на удаленном сервере. Основные процессы. Понятие тестового и рабочего программных окружений.

### Управление конфигурациями. Основные понятия, назначение. Примеры систем.

**Конфигурационное управление** — комплекс методов, направленных на систематический учёт изменений, вносимых разработчиками в [программный продукт](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82) в процессе его [разработки](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) и [сопровождения](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BF%D1%80%D0%BE%D0%B2%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), сохранение целостности системы после изменений, предотвращение нежелательных и непредсказуемых эффектов, формализацию процесса внесения изменений.

В целом, конфигурационное управление отвечает на вопрос: «Кто-то уже сделал нечто, как нам это воспроизвести?»

Конфигурацией можно управлять, контролируя документы, описывающие конечный продукт, требования к нему, всю его [проектную](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%BD%D0%B0%D1%8F_%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F) и [технологическую документацию](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%85%D0%BD%D0%BE%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F).

В связи с высокой динамичностью сферы разработки ПО, в ней конфигурационное управление особенно полезно. К процедурам можно отнести создание [резервных копий](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B7%D0%B5%D1%80%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BA%D0%BE%D0%BF%D0%B8%D1%8F), [контроль исходного кода](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F%D0%BC%D0%B8), [требований](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7_%D1%82%D1%80%D0%B5%D0%B1%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B9_(%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%9F%D0%9E)) проекта, [документации](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F_%D0%BD%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) и т. д.

Цели конфигурационного управления:

* Контроль: SCM позволяет отслеживать изменения в контролируемых объектах, обеспечивает соблюдение процесса разработки;
* Управление: SCM диктует процесс автоматической идентификации в ходе всего [жизненного цикла ПО](https://ru.wikipedia.org/wiki/%D0%96%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%86%D0%B8%D0%BA%D0%BB_%D0%9F%D0%9E), обеспечивает простоту модификации и сопровождения ПО;
* Качество;

Задачи конфигурационного управления:

* идентификация конфигурации;
* контроль конфигурации: контроль над изменениями материалов;
* учёт текущего состояния: состояние документов, состояние кода, состояние отдельных задач и всего проекта в целом;
* управление процессом разработки;
* управление сборкой;
* управление окружением;
* отслеживание задач и проблем (в частности, [отслеживание ошибок](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BE%D1%82%D1%81%D0%BB%D0%B5%D0%B6%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BE%D1%88%D0%B8%D0%B1%D0%BE%D0%BA));

**Процедуры управления конфигурацией** -

*Ревизия конфигурации* — процесс проверки соответствия документа нижнего уровня всем требованиям верхнего.

[*Аудит*](https://ru.wikipedia.org/wiki/%D0%90%D1%83%D0%B4%D0%B8%D1%82) *конфигурации* — процесс проверки соответствия готового продукта или его части документации.

*Контроль конфигурации* — процесс, при котором все предлагаемые изменения продукта проходят одобрение специальной группы (или отдельного человека). Одна из функций такой группы — контроль актуальности всех имеющихся документов, а также контроль того что все изменения сначала вносятся в документацию, а уже затем в объект изменения.

*Учет состояния конфигурации* — процесс подготовки отчетов о текущем состоянии продукта и состоянии утверждённых изменений.

Примеры систем - FreshService, ServiceNow, Samanage, [Ansible](https://en.wikipedia.org/wiki/Ansible_(software)), [Bcfg2](https://en.wikipedia.org/wiki/Bcfg2), [CFEngine](https://en.wikipedia.org/wiki/CFEngine), [Chef](https://en.wikipedia.org/wiki/Chef_(software)), [Nix](https://en.wikipedia.org/wiki/Nix_package_manager), [Otter](https://en.wikipedia.org/wiki/Otter_(software)), [Puppet](https://en.wikipedia.org/wiki/Puppet_(software)), [Quattor](https://en.wikipedia.org/wiki/Quattor), [SaltStack](https://en.wikipedia.org/wiki/SaltStack), [Terraform](https://en.wikipedia.org/wiki/Terraform_(software)), [Pulumi](https://en.wikipedia.org/w/index.php?title=Pulumi_(software)&action=edit&redlink=1) и [Vagrant](https://en.wikipedia.org/wiki/Vagrant_(software)).

### Контейнеризация программных приложений. Основные понятия, использование.

Контейнеризация — это подход к разработке программного обеспечения, при котором приложение или служба, их зависимости и конфигурация (абстрактные файлы манифеста развертывания) упаковываются вместе в образ контейнера. Контейнерное приложение может тестироваться как единое целое и развертываться как экземпляр образа контейнера в операционной системе (ОС) узла. Контейнеры также изолируют приложения друг от друга в общей операционной системе. Контейнерные приложения выполняются на основе узла контейнеров, который в свою очередь работает в операционной системе (Linux или Windows). Поэтому контейнеры требуют гораздо меньше ресурсов, чем образы виртуальных машин.

Каждый контейнер может вмещать целое веб-приложение или службу.

Еще одним преимуществом контейнеризации является масштабируемость. Вы можете быстро осуществлять горизонтальное масштабирование, создавая контейнеры для краткосрочных задач. С точки зрения приложения, создание экземпляра образа (создание контейнера) аналогично созданию экземпляра процесса, например, для службы или веб-приложения. Но для обеспечения надежности при запуске нескольких экземпляров одного образа на нескольких серверах обычно желательно, чтобы контейнеры (экземпляры образа) выполнялись на разных серверах или виртуальных машинах в разных доменах.

Иными словами, контейнеры предоставляют такие преимущества, как изоляция, переносимость, гибкость, масштабируемость и контроль, на протяжении всего жизненного цикла приложения. Самым важным преимуществом является изоляция среды разработки от рабочей среды.сбоя.

### **Примерные практические задания к экзамену**

1. Напишите программу, которая создает нить. Родительская и вновь созданная нити должны распечатать десять строк текста.
2. Напишите простой эхо-сервер, использующий неблокирующие сокеты и клиент к нему.
3. Напишите простой многопоточный загрузчик URL. Список URL скрипт принимает как аргументы командной строки.
4. Реализуйте простой HTTP-клиент. Он принимает один параметр командной строки - URL. Клиент делает запрос по указанному URL и выдает тело ответа на терминал как текст.
5. Напишите программу, которая вычисляет число Пи при помощи ряда Эйлера. Количество потоков программы должно определяться параметром командной строки.
6. Дана функция calculate(x, y). Напишите программу, которая создает пул из 5 процессов и распределяет в этом пуле вычисление функции на промежутке х от 0 до 1 с шагом 0,1. у равняется 2 всегда.
7. Напишите программу, которая проверяет все числа от 0 на простоту и выводит простые числа на экран по мере нахождения. Числа должны проверяться в различных потоках (или процессах, по выбору студента) Программа должна работать до тех пор, пока ее не остановит пользователь.
8. Напишите программу, которая обходит все файлы в директории, переданной ей как параметр и выводит на экран имена тех, чей размер задан как второй параметр. Реализовать рекурсивный обход поддиректорий.
9. Напишите программу, которая выводит на экран список номеров открытых портов на данной машине. Использовать команду netstat.
10. Напишите программу, которая копирует файл с удаленного хоста в текущую папку по SSH. Имя файла и адрес хоста принимать как параметры.