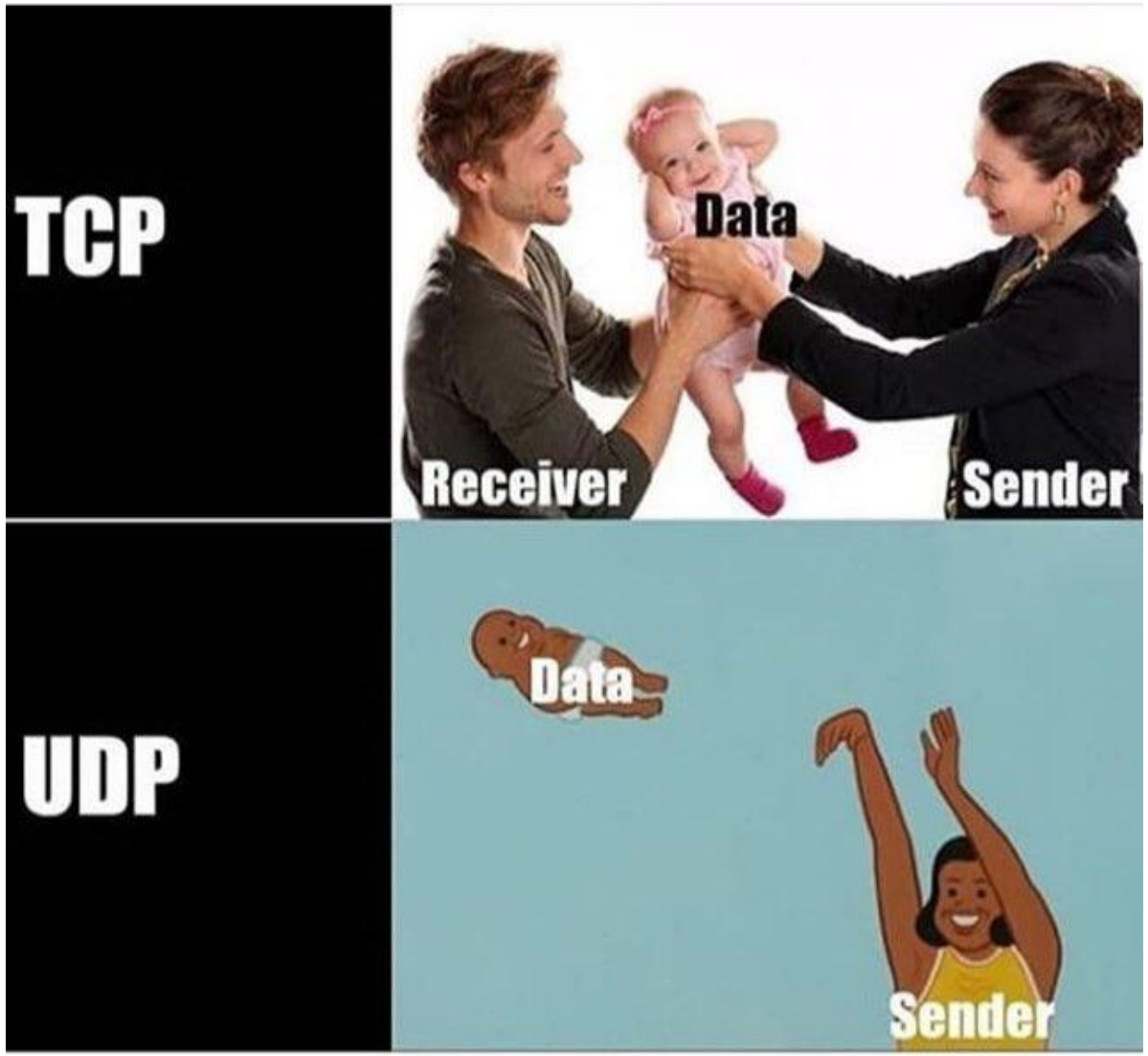
Сервис-ориентированные архитектуры

Связь и обмен данными в глобальных сетях

Глеб Игоревич Радченко

30.01.2021

Организация связи между процессами в глобальных сетях



Стек протоколов OSI

Application

Presentation

Session

Transport

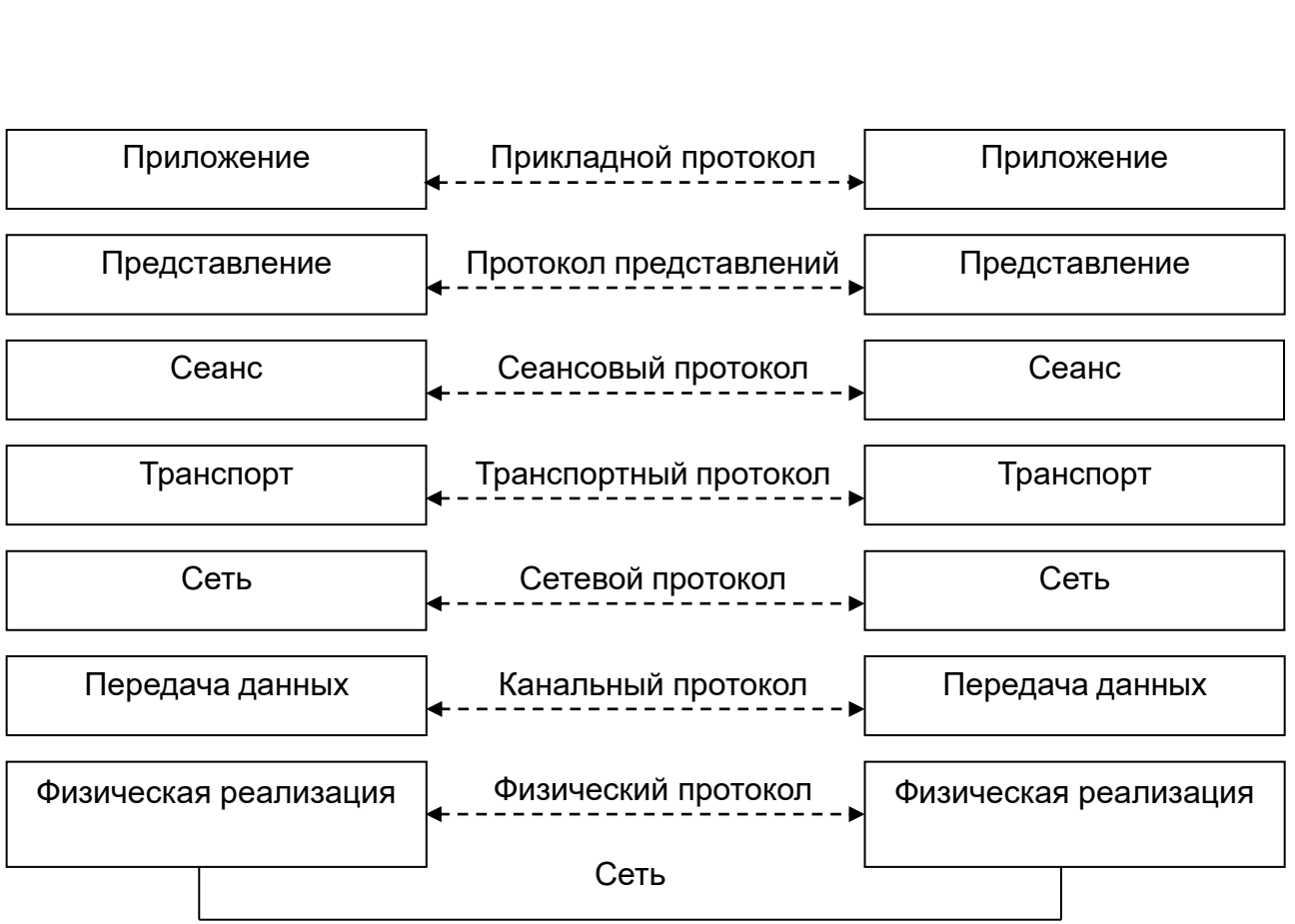
Network

Data Link

Physical

Стек протоколов OSI

OSI: Open Systems Interconnection model



FTP, SMTP, HTTP

XML, JPEG, ASCII, сжатие

PPTP, SMPP

TCP, UDP

IP, IPsec

IEEE 802.11, PPP, PPPoE

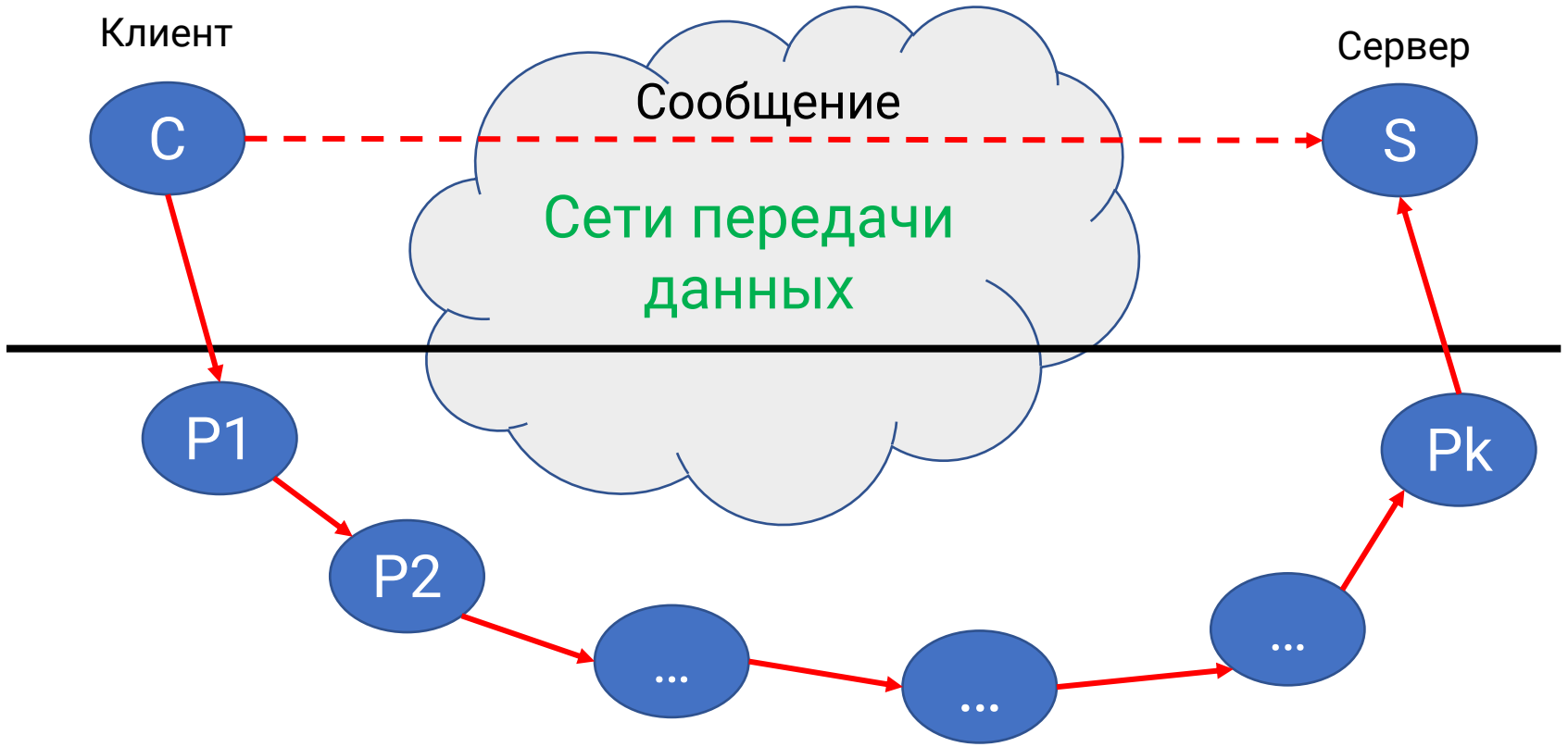
Bluetooth, DSL, IRDA

Стек протоколов OSI

- *Прикладной (application) протокол*: конкретные потребности пользовательских приложений. Примерами являются электронная почта, доски объявлений, чаты, веб-приложения, службы каталогов и т.д.
- *Протокол представления (presentation)*: решает проблемы совместимости, устраняя синтаксические различия в представлении данных.
- *Сеансовый (session) протокол:* управляет созданием/завершением сеанса, обменом информацией, синхронизацией задач
- *Транспортный (transport) протокол:* обеспечивает сквозное соединение между отправителем и приемником.
- *Сетевой (network) протокол:* обеспечивает межмашинная коммуникацию (связь машина-к-машине), и несет ответственность за маршрутизации сообщений.
- *Канальный (data-link) протокол:* собирает поток битов в фреймы и добавляет управляющие биты для защиты данных от порчи в процессе передачи.
- *Физический (physical) протокол:* определяет, каким образом биты передаются по каналу связи. В электрической связи, определяет уровни напряжения (или частоты), которые будут использоваться для представления 0 или 1.

Транспортный и сетевой уровень

Транспортный протокол и выше

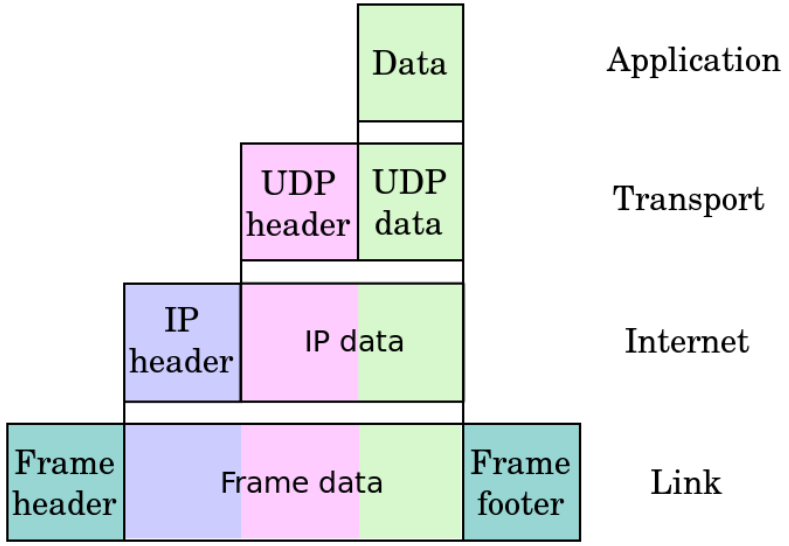


Сетевой протокол и ниже

TCP/IP

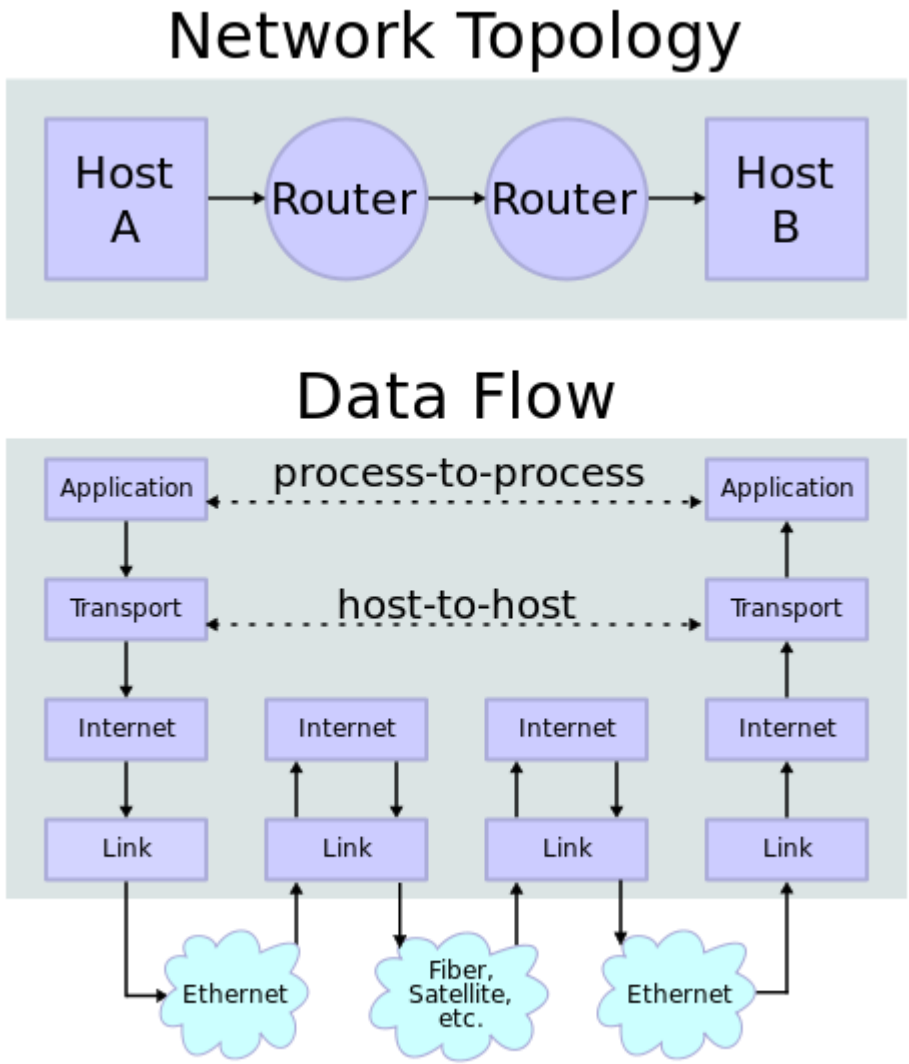
- Наиболее популярный стек протоколов в сети Интернет
- Четыре слоя

Прикладной слой	SMTP, HTTP, Telnet, FTP и др.
Транспортный слой	UDP, TCP
Сетевой слой	IP
Слой связи	Ethernet, WiFi



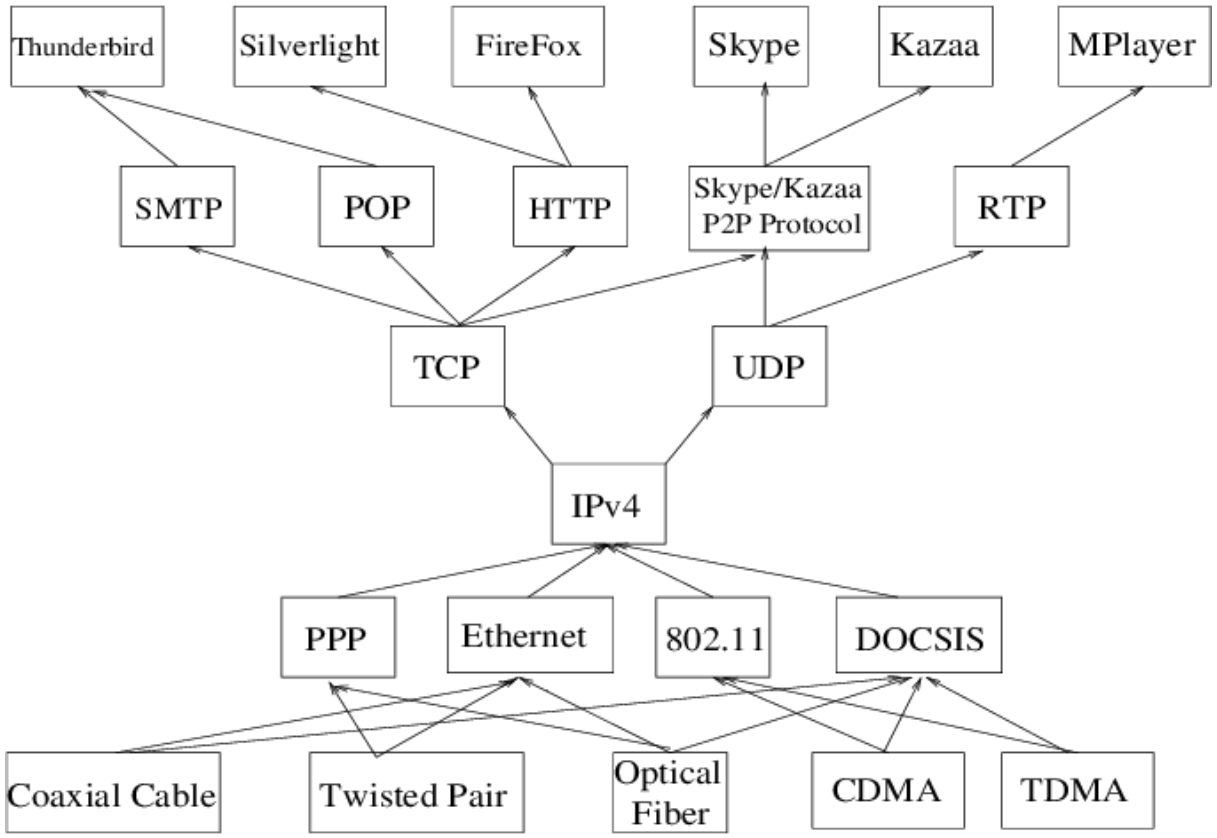
Протокол IP

- Определяет датаграмму как единицу передачи данных
- Определяет схему интернет-адреса
- Обеспечивает передачу датаграмм от отправителя к получателю



Песочные часы стека сетевых протоколов

- Всё на базе IP, IP на базе всего
- Все приложения базируются на IP
- IP может обеспечить коммуникацию на базе любой физической сети
- IP – в сердце любой сетевой коммуникации



IP-адрес

- Уникальный сетевой адрес узла в компьютерной сети, построенной на основе стека протоколов TCP/IP.
- В сети Интернет требуется глобальная уникальность адреса; в случае работы в локальной сети требуется уникальность адреса в пределах сети.

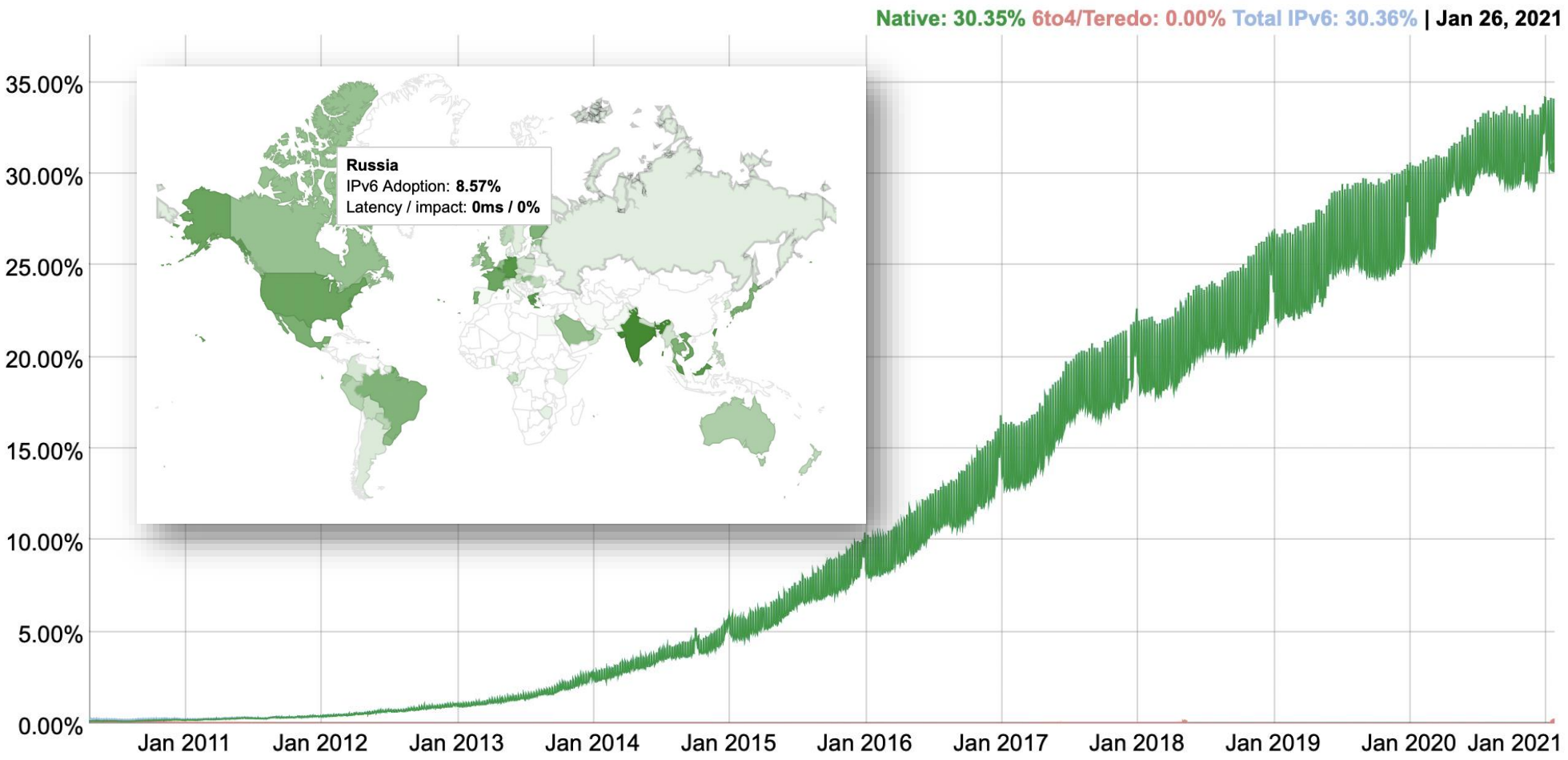
- IPv4 => длина 4 байта, например

  192.168.0.3

- IPv6 => длина 16 байт, например

  2001:0db8:85a3:0000:0000:8a2e:0370:7334

IPv6 в мире


Native: 30.35% 6to4/Teredo: 0.00% Total IPv6: 30.36% | Jan 26, 2021
Russia
IPv6 Adoption: 8.57%
Latency / impact: 0ms / 0%
35.00%
30.00%
25.00%
20.00%
15.00%
10.00%
5.00%
0.00%
Jan 2011
Jan 2012
Jan 2013
Jan 2014
Jan 2015
Jan 2016
Jan 2017
Jan 2018
Jan 2019
Jan 2020
Jan 2021

IP адреса и доменные адреса

- Одно доменное имя может преобразовываться поочерёдно в несколько IP-адресов (например, для распределения нагрузки).
- Одновременно, один IP-адрес может использоваться для множества доменных имён различных сайтов (тогда при доступе они различаются по доменному имени)
- Также, сервер с одним доменным именем может содержать несколько разных сайтов, а части одного сайта могут быть доступны по разным доменным именам

Заголовок IP-пакета

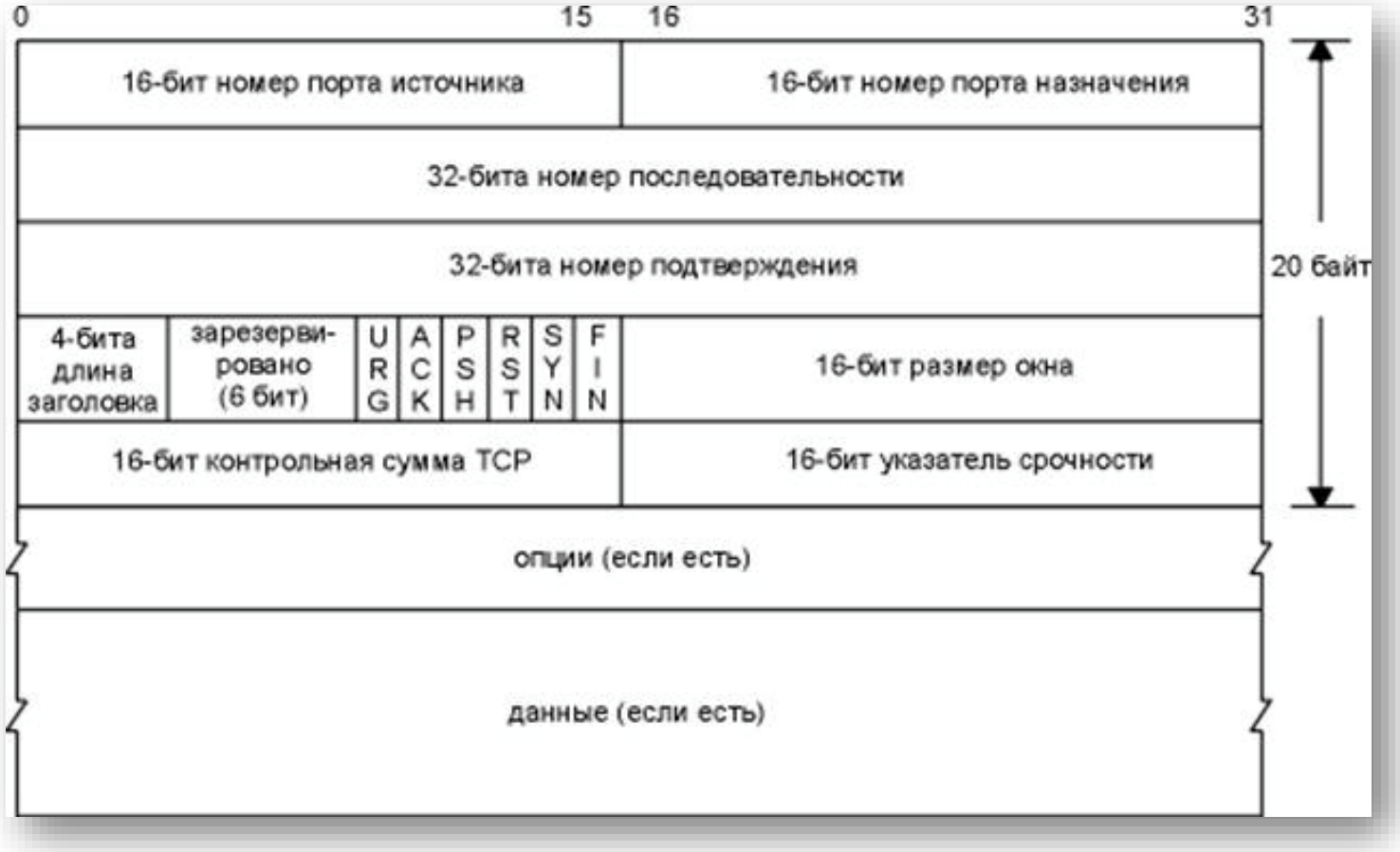
<table>
<tr><td>4 бита
Номер
версии</td><td>4 бита
Длина
заголовка</td><td>8 бит
Тип
сервиса</td><td colspan="2">16 бит
Общая длина</td></tr>
<tr><td colspan="3">16 бит
Идентификатор пакета</td><td>3 бита
Флаги</td><td>13 бит
Смещение фрагмента</td></tr>
<tr><td colspan="2">8 бит
Время жизни</td><td>8 бит
Тип протокола</td><td colspan="2">16 бит
Контрольная сумма</td></tr>
<tr><td colspan="5">32 бита
IP-адрес отправителя</td></tr>
<tr><td colspan="5">32 бита
IP-адрес получателя</td></tr>
<tr><td colspan="5">Опции и выравнивание
(не обязательно)</td></tr>
</table>

Протоколы TCP и UDP

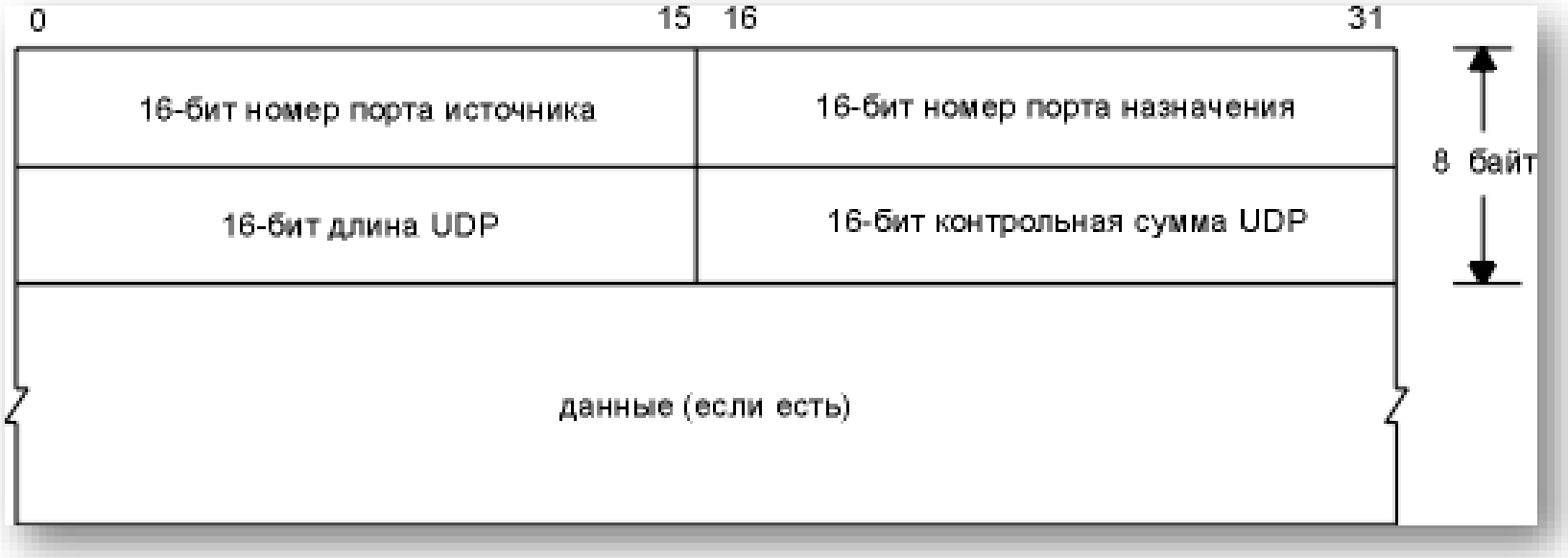
- TCP/IP – транспортный слой, обеспечивающий передачу данных от клиента – серверу и наоборот.
- 2 ключевых протокола: TCP и UDP.

Слой	TCP	UDP
Прикладной	Данные передаются в потоках	Данные передаются в сообщениях
Транспортный	Сегмент	Пакет
Сетевой	Датаграмма	Датаграмма
Слой связи	Фрейм	Фрейм

Заголовок TCP



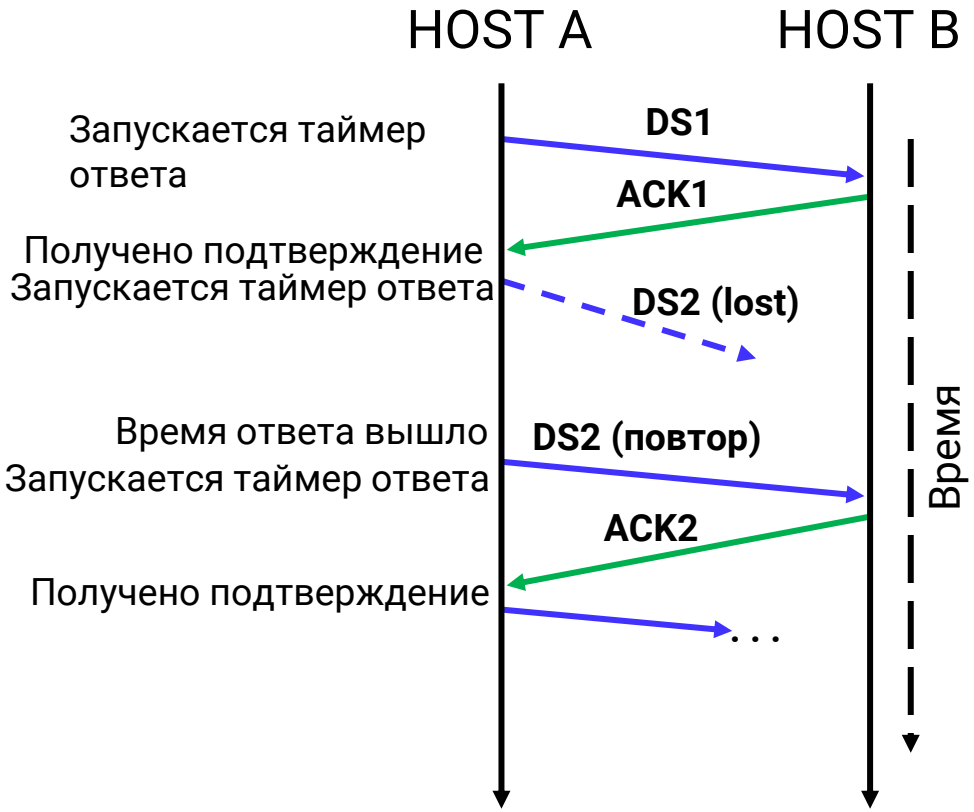
Заголовок UDP



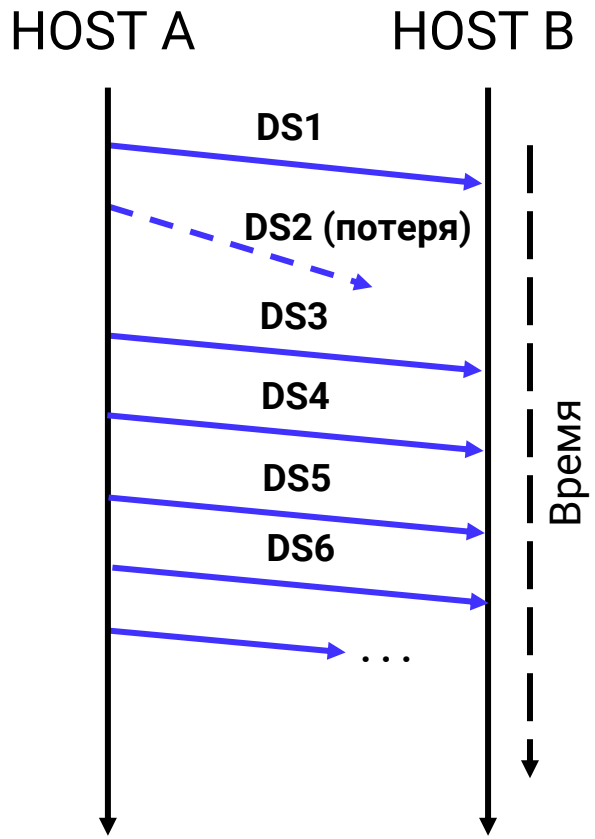
TCP vs UDP

Передача данных TCP

ACK – подтверждение
DS – поток данных (Data Stream)



Передача данных UDP



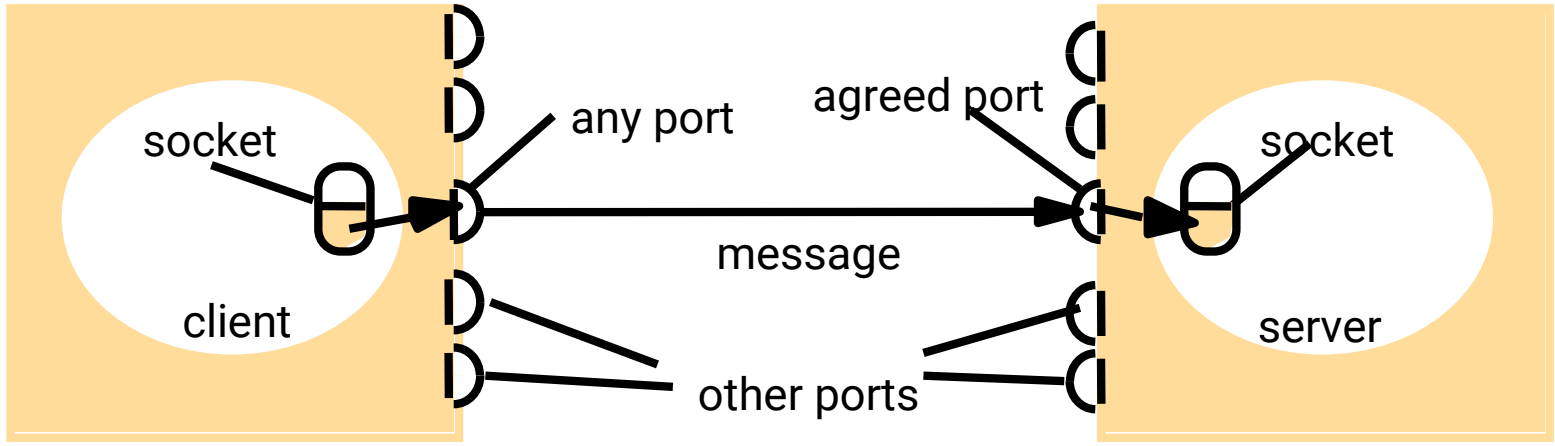
Организация связи между удаленными системами: сокеты

Интерфейс сокета Беркли

- Реализация интерфейса лежит в основе TCP/IP, благодаря чему считается одной из фундаментальных технологий, на которых основывается Интернет.
- Все современные операционные системы имеют ту или иную реализацию интерфейса сокетов Беркли, так как это стало стандартным интерфейсом для подключения к сети Интернет.

Прямая передача сообщений: сокеты

- Т.е. используется непосредственно транспортный уровень в виде Middleware.



- **Сокет –** абстрактный объект, представляющий конечную точку соединения, обеспечивающий прием и передачу сообщений внешнему (локальному или удаленному) процессу.
- **Сокет TCP/IP –** комбинация IP-адреса и номера порта, например 10.10.10.10:80.
- Интерфейс сокетов впервые появился в BSD Unix.

Berkeley Sockets API

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

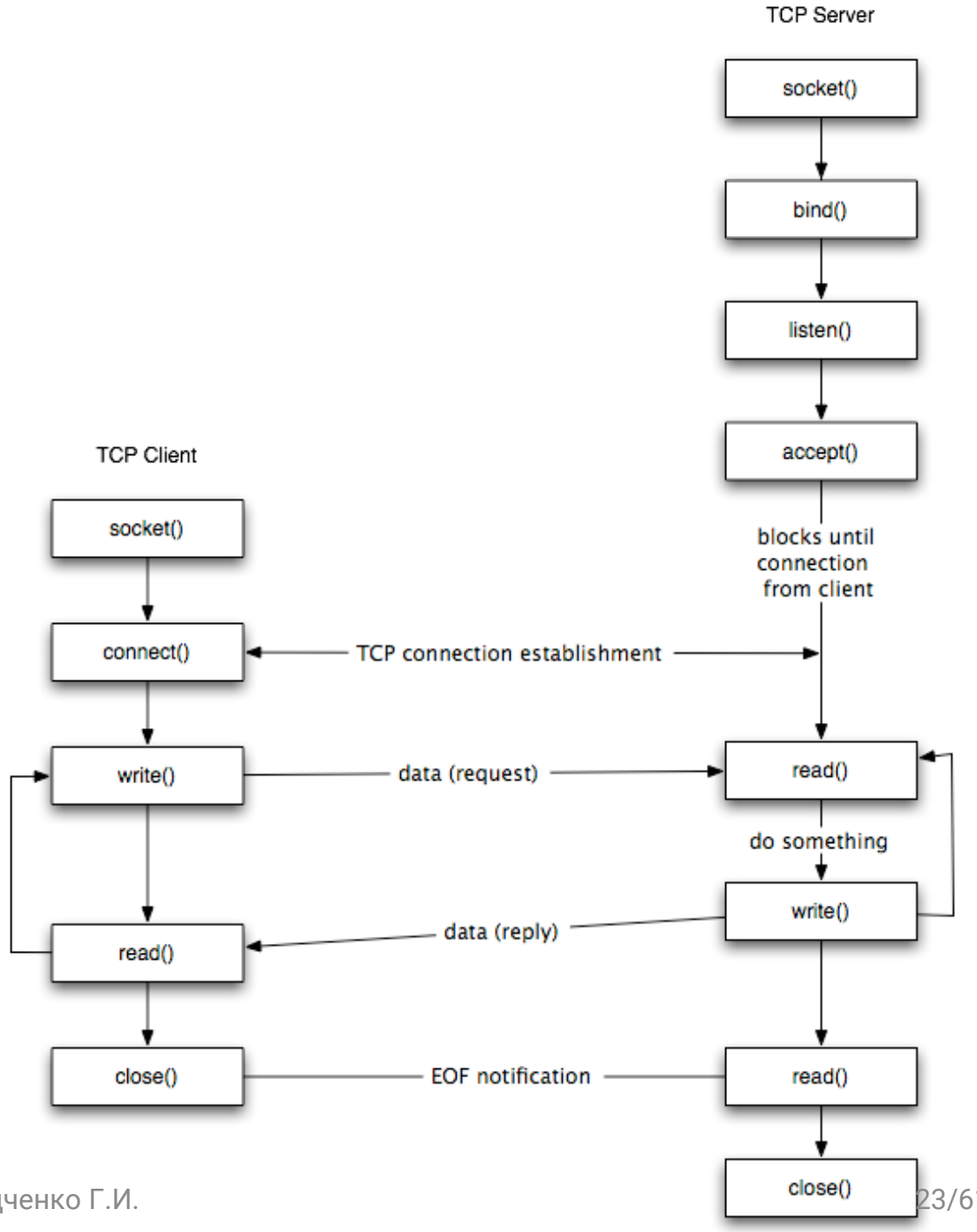
Socket primitives for TCP/IP.

Специфика TCP-сервера

TCP реализует концепцию соединения.

- Процесс создаёт TCP-сокет вызовом функции **socket()** с параметрами PF_INET (для IPv4) или PF_INET6 (для IPv6), типом SOCK_STREAM (Потоковый сокет)
- Далее происходит вызов **bind()** для связи сокета с парой адрес/порт
- Далее, происходит подготовка сокета к прослушиванию на предмет соединений (создание прослушиваемого сокета) при помощи вызова **listen()**
- Принятие входящих соединений происходит через блокирующий вызов **accept()**. После установления соединения, возвращается дескриптор сокета для принятого соединения
- Коммуникация производится операциями **read()** и **write()** в контексте открытого соединения

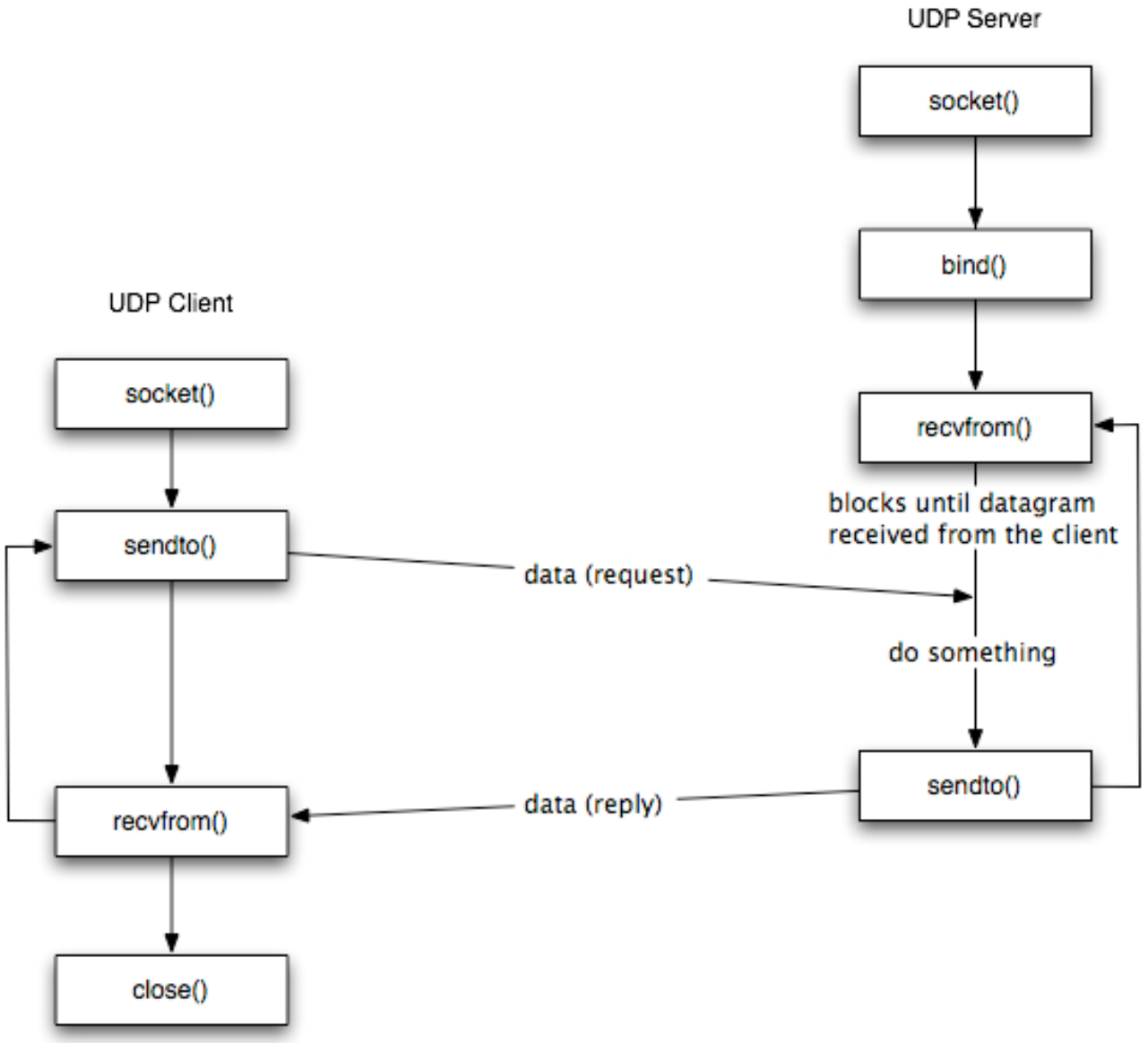
Berkeley Sockets (коммуникационная диаграмма TCP-соединения)



Специфика UDP-сервера

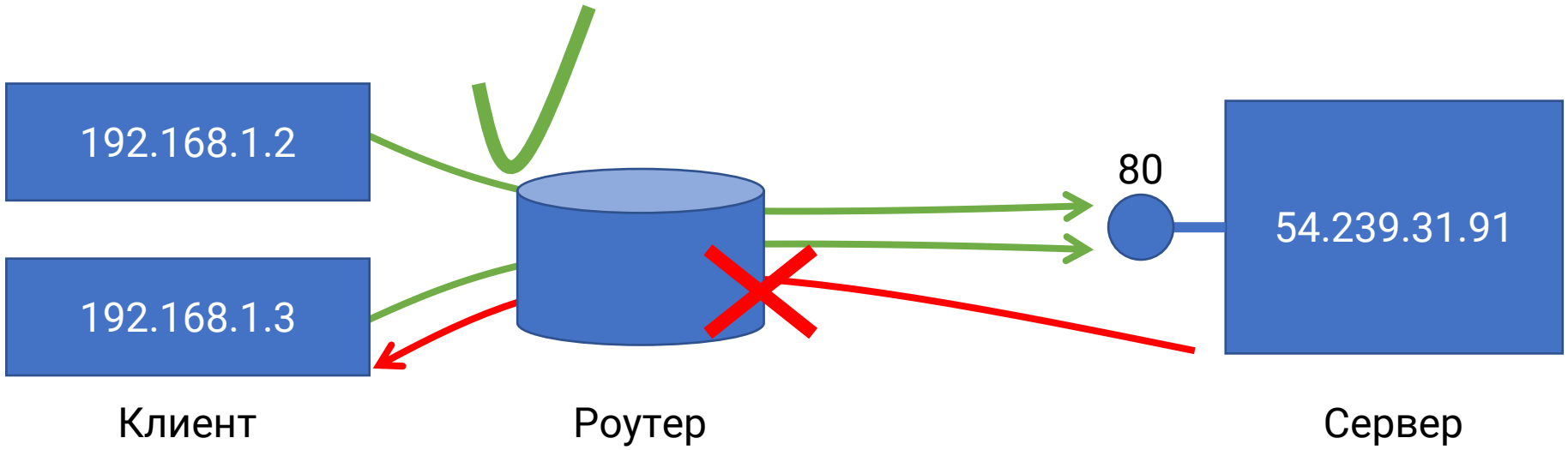
- UDP основывается на протоколе без установления соединений, то есть протокол, не гарантирует доставку информации. UDP-пакеты могут приходить не в указанном порядке, дублироваться и приходить более одного раза, или даже не доходить до адресата вовсе.
- Отсутствие установки соединений означает отсутствие потоков или соединений между двумя хостами, так как вместо этого данные прибывают в датаграммах.
- Процесс создаёт TCP-сокет вызовом функции **socket()** с параметрами PF_INET (для IPv4) или PF_INET6 (для IPv6), типом SOCK_DGRAM (Датаграмный сокет)
- Далее происходит вызов **bind()** для связи сокета с парой адрес/порт (область номеров UDP-портов полностью отделено от TCP-портов)
- Дальнейшее получение входящих данных происходит при помощи функции **recvfrom()** а отправка – при помощи функции **sendto().** При каждой отправке необходимо явно указывать адрес и порт получателя сообщения

Berkeley Sockets (коммуникационная диаграмма UDP-соединения)



Как обеспечивается доставка ответа?

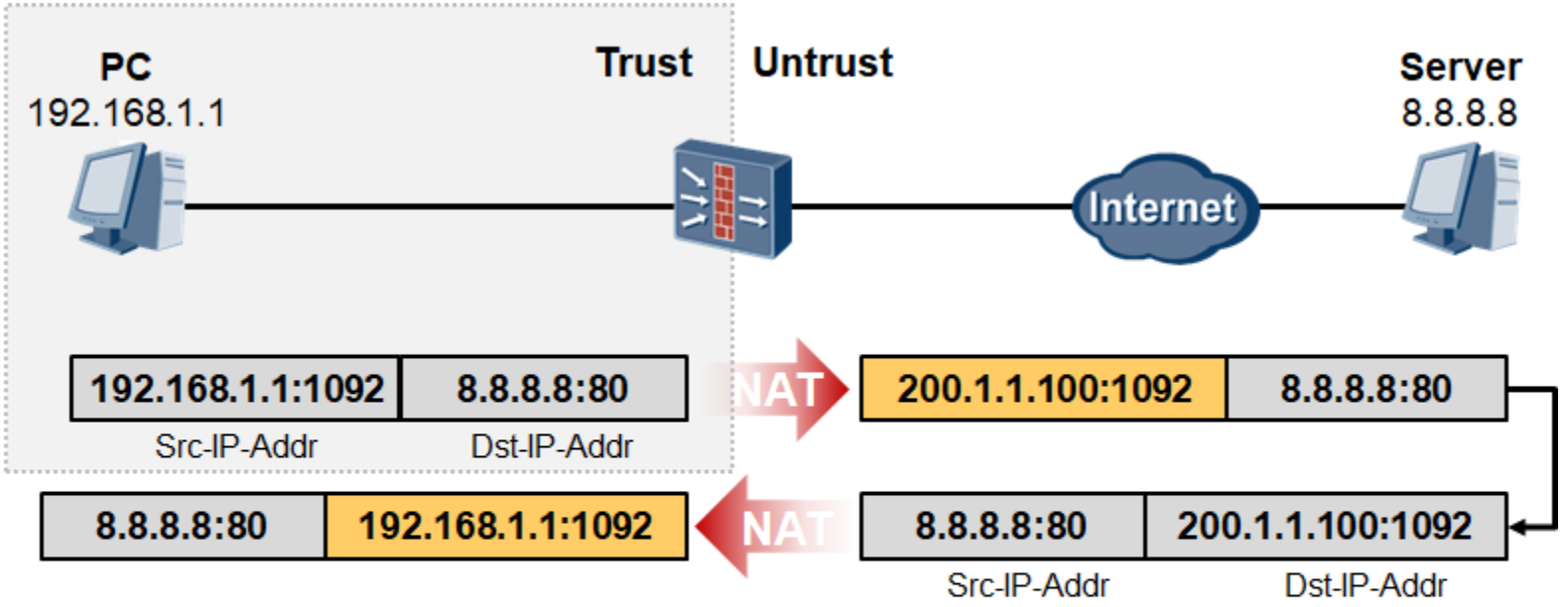
- Понятно, как сообщения от клиента доходят до сервера: у сервера есть публичный IP адрес, открытый порт, по которому прилетают сообщения от клиентов.
- Но как сообщения от сервера возвращаются назад?
- Клиент не имеет публичного IP-адреса, доменного имени, он сидит за роутером, который (по умолчанию) блочит все запросы на входящие соединения…



PAT - Port Address Translation

- Клиент также открывает и использует уникальный номер порта (может быть выделен из «высокого» диапазона самой ОС, если не определен явно) для отправки сообщений.
- Когда клиент инициирует соединение с сервером, роутер:
  1. запоминает, с какого IP адреса и порта произошла установка соединения
  2. заменяет IP адрес клиента на свой внешний IP, порт клиента на собственный уникальный порт и направляет пакет дальше
  3. как только от сервера приходит ответ на порт роутера, он заменяет IP адрес на адрес клиента, порт – на тот порт, с которого произошла установка соединения и отправляет пакет клиенту

PAT - Port Address Translation



NAT mapping on Firewall

Inside	Global
192.168.1.1	200.1.1.100

Сколько держится порт на роутере?

- При установке **TCP** соединения PAT поддерживается вплоть до явного закрытия соединения (обмен сообщениями FIN/FIN-ACK). Если такого обмена не происходит, то PAT должен поддерживаться порядка **2-х часов** (RFC 5382: https://tools.ietf.org/html/rfc5382), т.к. TCP соединения могут быть открыты «бесконечно долго» без явного обмена сообщениями.
- При отправке **UDP** запроса PAT поддерживается порядка **2-х минут** (RFC 4787: https://tools.ietf.org/html/rfc4787), хотя ряд роутеров могут устанавливать и меньшие задержки (так, OpenWRT по умолчанию держит порт порядка 1 минуты). Таким образом, для поддержания стабильного соединения по UDP необходимо обеспечивать периодические «пинги» сервера, если со стороны клиента не отправляется другой информации

Примеры приложения с использованием сокетов

- При желании, вы можете изучить ряд приложений-примеров, иллюстрирующих базовые аспекты установления соединения и обмена данными между клиентом и сервером посредством BSD Sockets:
  - Крестики-нолики на Go: https://github.com/domage/tictactoe
  - Голосовой чат на Python (иллюстрирует как связь на базе TCP, так и на базе UDP): https://github.com/domage/soa-curriculum-2021/tree/main/examples/sockets-voice-chat

Пример: протокол общения

- Сообщения – текстовые строки формата ASCII
- 2 типа сообщений: команды и ходы
- Команды:

```
const (
    StopCommand = "Exit" – завершение игры
    BoardCommand = "Board" – состояние поля
    InitCommand = "Init" – подключение к игре
    TurnRequest = "Turn" – информация – чей ход
)
```

- Ходы:
  - Координата по вертикали (от 0 до 2), координата по горизонтали (от 0 до 2), крестик или нолик (X или 0):
    - 00X
    - 220

Функция main

```
func main() {
    connect := flag.String("connect", "", "IP address of server")
    flag.Parse()

    // If the connect flag is set, go into client mode.
    if *connect != "" {
        err := sockets.StartClient()
        log.Println("Client done.")
        return
    }

    // Else go into server mode.
    err := sockets.StartServer()
}
```

Клиент: запуск

```
func StartClient() error {

    var (
        ip = "127.0.0.1"
        port = 3333
    )

    return SocketClient(ip, port)
}
```

Работа клиента (упрощенно)

```
func SocketClient(ip string, port int) error {
    addr := strings.Join([]string{ip, strconv.Itoa(port)}, ":")
    conn, err := net.Dial("tcp", addr)

    defer conn.Close()
    ...

    buff := make([]byte, 1024)
    var s string
    for {
        fmt.Scanf("%s", &s)
        _, err = conn.Write([]byte(s))

        if s == StopCommand {
            return nil
        }

        buff = make([]byte, 1024)
        n, err = conn.Read(buff)
        if strings.HasSuffix(string(buff[:n]), StopCommand) {
            return nil
}}}
```

Сервер: запуск

```
func StartServer() error {
    port := 3333
    players = 0
    status = InitPhase
    err := socketServer(port)
    return err
}
```

Сервер: обработка подключений (упрощенно)

```
func socketServer(port int) error {
    listen, err := net.Listen("tcp4", ":"+strconv.Itoa(port))
    defer listen.Close()
    for {
        conn, err := listen.Accept()
        switch {
        case players == 0:
            status = InitPhase
            g = game.NewGame()
            players++
            go handler(conn)
            continue
        case players == 1 && status == InitPhase:
            status = GamePhase
            players++
            go handler(conn)
            continue
        case status == GamePhase || status == ExitPhase:
            conn.Close()
            log.Printf("The game is not finished yet")
}}}
```

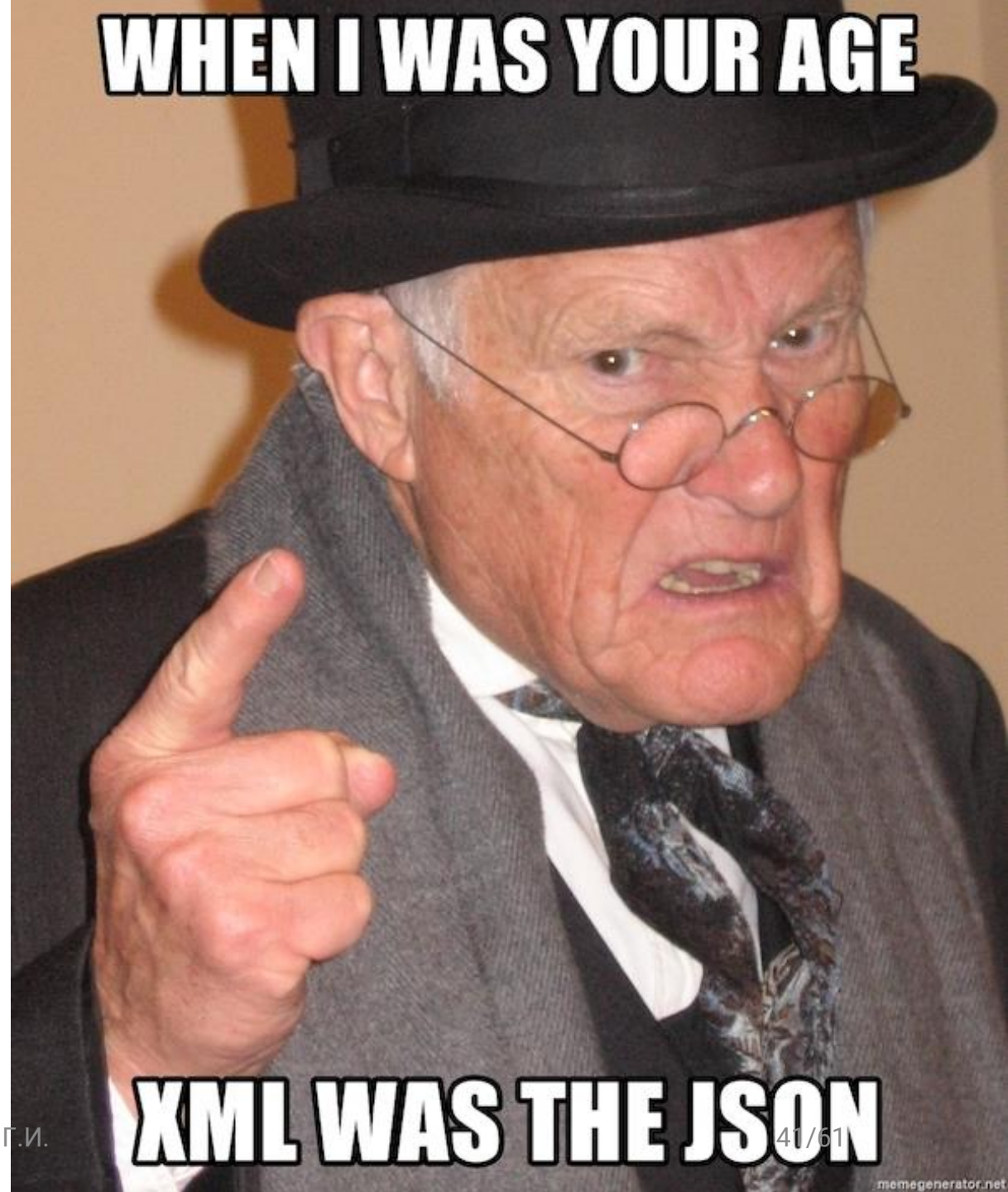
Сервер: обслуживание входящих запросов (1)

```
func handler(conn net.Conn) {
   defer conn.Close()
   var (
      buf = make([]byte, 1024)
      r = bufio.NewReader(conn)
      w = bufio.NewWriter(conn)
   )

ILOOP:
   for {
      n, err := r.Read(buf)
      data := string(buf[:n])
      switch err {
      case io.EOF:
         players--
         status = ExitPhase
         break ILOOP

      case nil:

         log.Println("Receive:", data)
         x, y, turn, parsingError := decode(data)

      switch {
```

Сервер: обработка команд

```
case strings.HasPrefix(data, InitCommand):
    var turn string
    if players == 1 { turn = "X" }
    else { turn = "0" }
    w.Write([]byte(turn))
    w.Flush()

case strings.HasPrefix(data, StopCommand):
    players--
    status = ExitPhase
    break ILOOP
case strings.HasPrefix(data, BoardCommand):
     board := fmt.Sprint(g)
     w.Write([]byte(board))
     w.Flush()
```

Форматы сериализации данных



Сериализация

- **Сериализация данных** - это процесс преобразования структур данных в последовательности бит, которые могут быть сохранены. Интуитивно, процесс десериализации создает соответствующую структуру данных из заданной последовательности бит.

Форматы сериализации данных

- Форматы сериализации данных: выходной формат (двоичный или текстовый) и наличие схемы данных.
- Схемы используются для обеспечения целостности данных, передаваемых между программным обеспечением, разработанным разными поставщиками или написанным на разных языках программирования.

	Без схемы	Со схемой
Текстовый	• XML • JSON • YAML	• XML+XSD • JSON+JSON Schema
Двоичный	• MessagePack • BSON	• Google Protocol Buffers • Apache Thrift • Apache Avro

XML vs JSON

XML

```
<person>
  <firstName>Иван</firstName>
  <lastName>Иванов</lastName>
  <address>
    <streetAddress>Московское ш., 101,
кв.101</streetAddress>
    <city>Ленинград</city>
    <postalCode>101101</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

291 байт

JSON

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address":  {
      "streetAddress": "Московское ш.,
101, кв.101",
      "city": "Ленинград",
      "postalCode": 101101
   },
   "phoneNumbers":  [
      "812 123-1234",
      "916 123-4567"
   ]
}
```

174 байта

MessagePack

- MessagePack – это бинарный формат предоставления данных, структура которого напоминает JSON (только более быстрый и более компактный).
- Был спроектирован, чтобы обеспечивать прозрачную конвертацию в JSON

JSON vs MessagePack

	JSON		MessagePack	
null	`null`	4 bytes	`c0`	1 byte
Integer	`10`	2 bytes	`0a`	1 byte
Array	`[20]`	4 bytes	`91 14`	2 bytes
String	`"30"`	4 bytes	`a2 '3'`	3 bytes
Map	`{"40":n`	11 bytes	`81 a1 '4'`	5 bytes

Architecture of MessagePack by Sadayuki Furuhashi
http://www.slideshare.net/frsyuki/architecture-of-messagepack

MessagePack

JSON

```
{
    "firstName": "Иван",
    "lastName": "Иванов",
    "address":  {
        "streetAddress": "Московское ш.,
101, кв.101",
        "city": "Ленинград",
        "postalCode": 101101
    },
    "phoneNumbers":  [
        "812 123-1234",
        "916 123-4567"
    ]
}
```

174 bytes

84 a9 66 69 72 73 74 4e 61 6d 65 a8 d0 98
d0 b2 d0 b0 d0 bd a8 6c 61 73 74 4e 61 6d
65 ac d0 98 d0 b2 d0 b0 d0 bd d0 be d0 b2
a7 61 64 64 72 65 73 73 83 ad 73 74 72 65
65 74 41 64 64 72 65 73 73 da 00 27 d0 9c
d0 be d1 81 d0 ba d0 be d0 b2 d1 81 d0 ba
d0 b
20 d0 ba d0 b2 2e 31 30 31 a4 63 69 74 79
b2 d0 9b
80 d0 b0 d0 b4 aa 70 6f 73 74 61 6c 43 6f
64 65 ce 00 01 8a ed ac 70 68 6f 6e 65 4e
75 6d 62 65 72 73 92 ac 38 31 32 20 31 32
33 2d 31 32 33 34 ac 39 31 36 20 31 32 33
2d 34 35 36 37

MessagePack (hex)
144 bytes 83 %

http://msgpack.org/

MessagePack

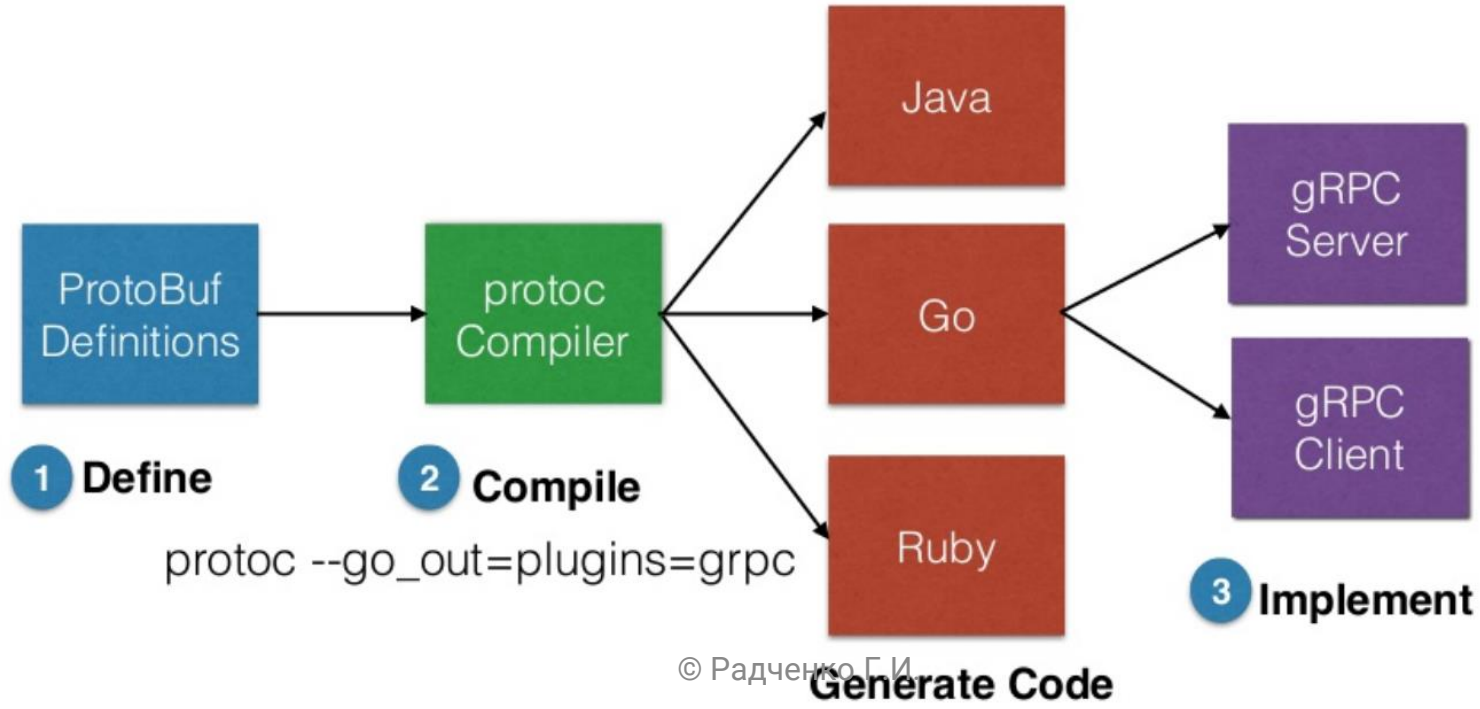
- **JSON+ZIP** VS **MessagePack**:
  - На 50% падает производительность при архивировании / разархивировании
  - Невозможно работать с данными в режиме потока (напрямую), необходима обязательная предварительная разархивация

Google Protocol Buffers

- Protocol Buffers — язык описания данных, предложенный Google в 2008 году, как альтернатива XML. Предполагается, что Protocol Buffers проще и легче, чем XML.
- Сначала должна быть описана структура данных, которая затем компилируется в классы, представляющие эти структуры. Вместе с классами идет код их сериализации в компактный формат представления.
- Примечательно, что можно добавлять к уже созданной структуре данных новые поля без потери совместимости с предыдущей версией: при анализе старых записей новые поля просто будут игнорироваться.
- Используется в большом количестве реальных коммерческих проектов (Diablo 3, Pokemon Go и др.)

Один прото-файл для многих

Компилятор прото-файлов доступен для языков Java, Python, Objective-C, C++, Go, JavaNano, Ruby, C#, и еще более 20 языков поддерживаются библиотеками сторонних разработчиков



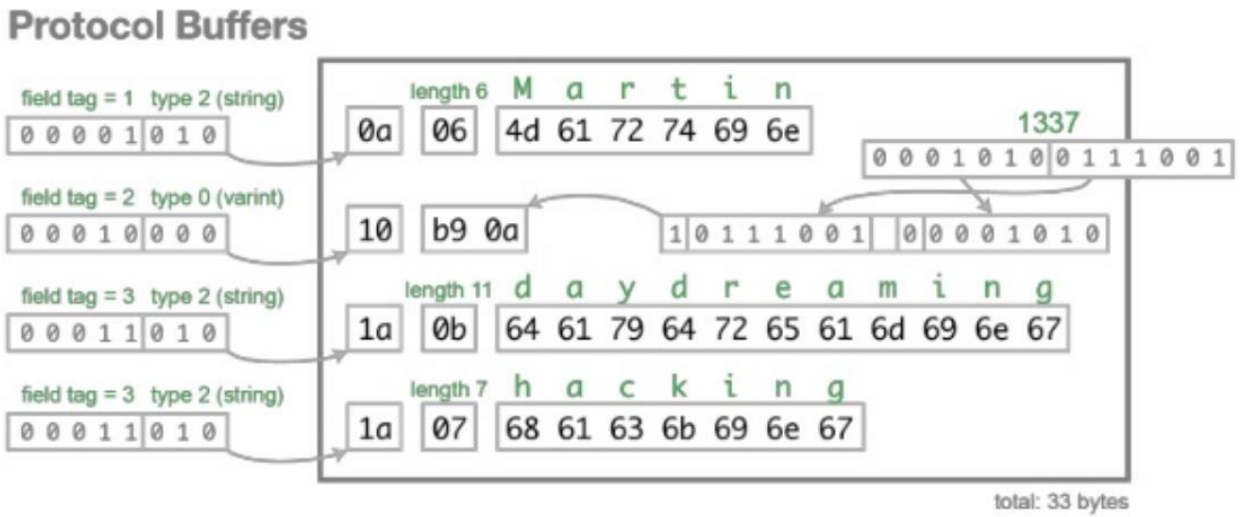
Формат proto-файла

```
message Car {
    required string model = 1;

    enum BodyType {
        sedan = 0;
        hatchback = 1;
        SUV = 2;
    }

    required BodyType type = 2 [default = sedan];
    optional string color = 3;
    required int32 year = 4;

    message Owner {
        required string name = 1;
        required string lastName = 2;
        required int64 driverLicense = 3;
    }

    repeated Owner previousOwner = 5;
}
```

.proto - файл

Person.json	Person.proto

{
"userName": "Martin",
"favouriteNumber": 1337,
"interests": ["daydreaming", "hacking"]
}
	message Person {
	required string user_name = 1;
	optional int64 favourite_number = 2;
	repeated string interests = 3;
	}



(c) Sergey Podolsky. https://www.slideshare.net/SergeyPodolsky/google-protocol-buffers-56085699

Apache Avro

- Apache Avro – это инфраструктура, которая позволяет сериализовать данные в формате со встроенной схемой.
- Сериализованные данные представлены в компактном двоичном формате, который не требует генерации прокси-объектов или программного кода.
- Вместо использования библиотек сгенерированных прокси-объектов и строгой типизации, формат Avro интенсивно задействует схемы, которые отсылаются вместе с сериализованными данными.
- Передача схем вместе с Avro-сообщениями позволяет любому приложению десериализовать данные.
- Очень активно применяется в системах работы с большими данными, таких как Apache Hadoop, Apache Spark и др.

Достоинства Apache Avro

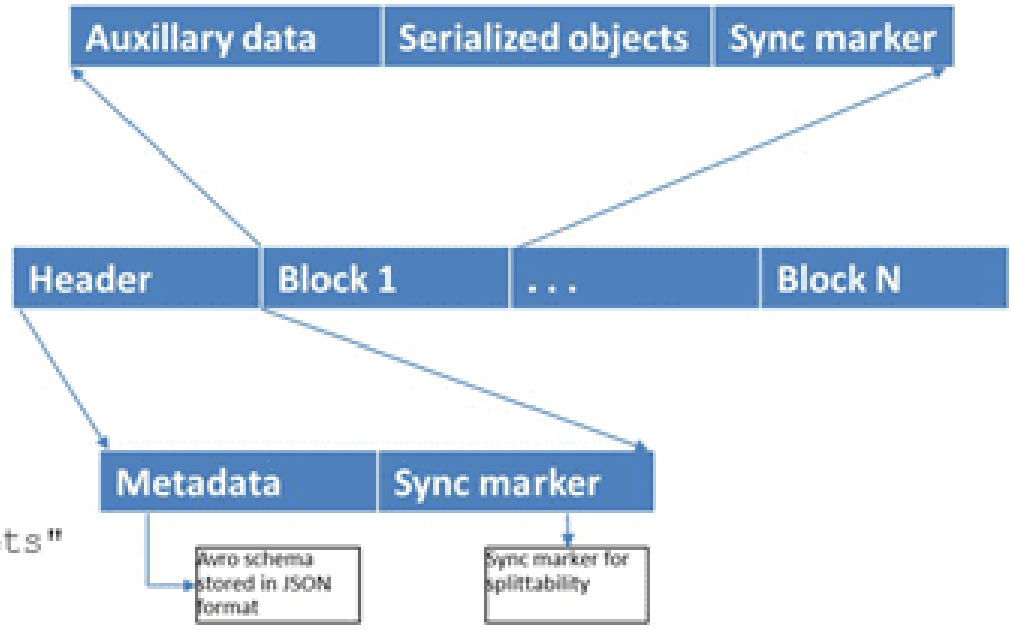
- Avro поддерживает эволюцию схем данных, обрабатывая изменения схемы путем пропуска, добавления или модификации отдельных полей.
- Независимая от реализации человекочитаемая JSON-схема данных обеспечивает поддержку множества языков программирования (C, C++, C#, Go, Haskell, Java, Python, Scala, другие скриптовые и ООП-языки), а также облегчает отладку в процессе разработки
- Большой набор библиотек, обеспечивающих генерацию схемы Avro непосредственно из исходного кода описания структур данных на большом числе языков программирования:
  - C# - Microsoft.Hadoop.Avro.AvroSerializer
  - Java - org.apache.avro.reflect
  - Scala - Avro4s
  - Python - AVRO-GEN

Пример описания схемы и структура файла Avro

Sample AVRO schema in JSON format

```
{
  "type" : "record",
  "name" : "tweets",
  "fields" : [ {
    "name" : "username",
    "type" : "string",
}, {
    "name" : "tweet",
    "type" : "string",
}, {
    "name" : "timestamp",
    "type" : "long",
} ],
  "doc:" : "schema for storing tweets"
}
```

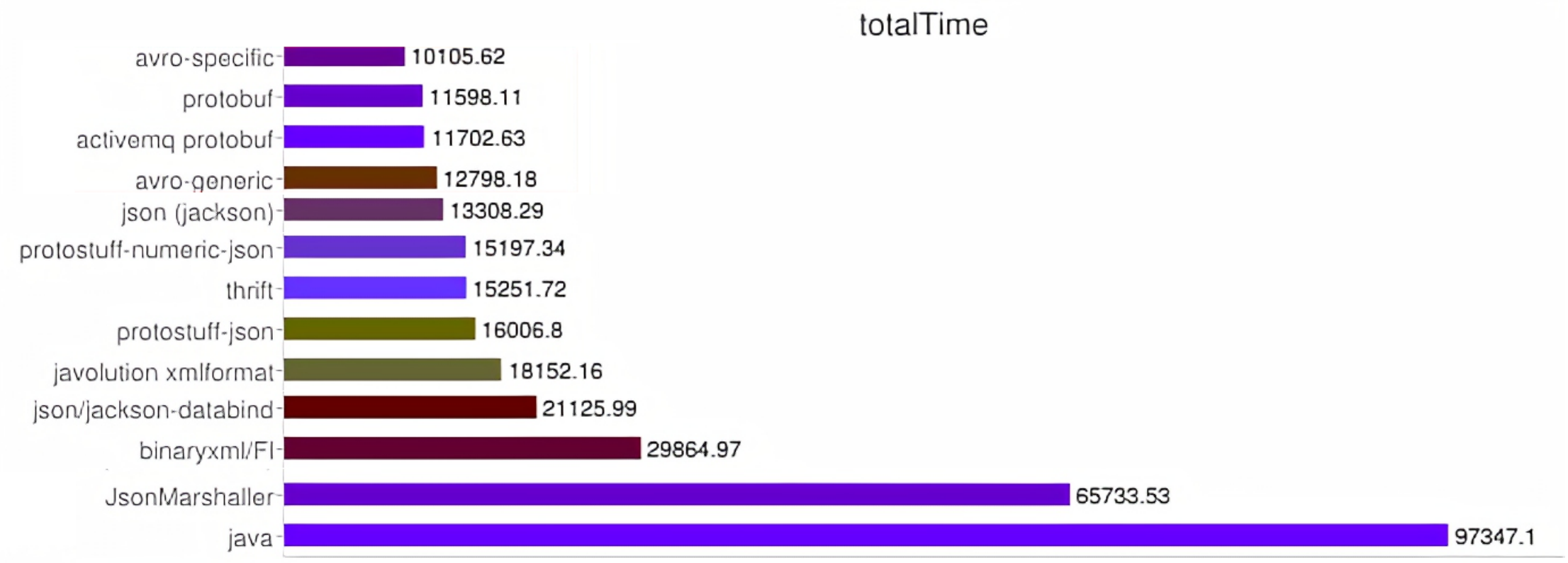
Avro file structure



Форматы обмена данными

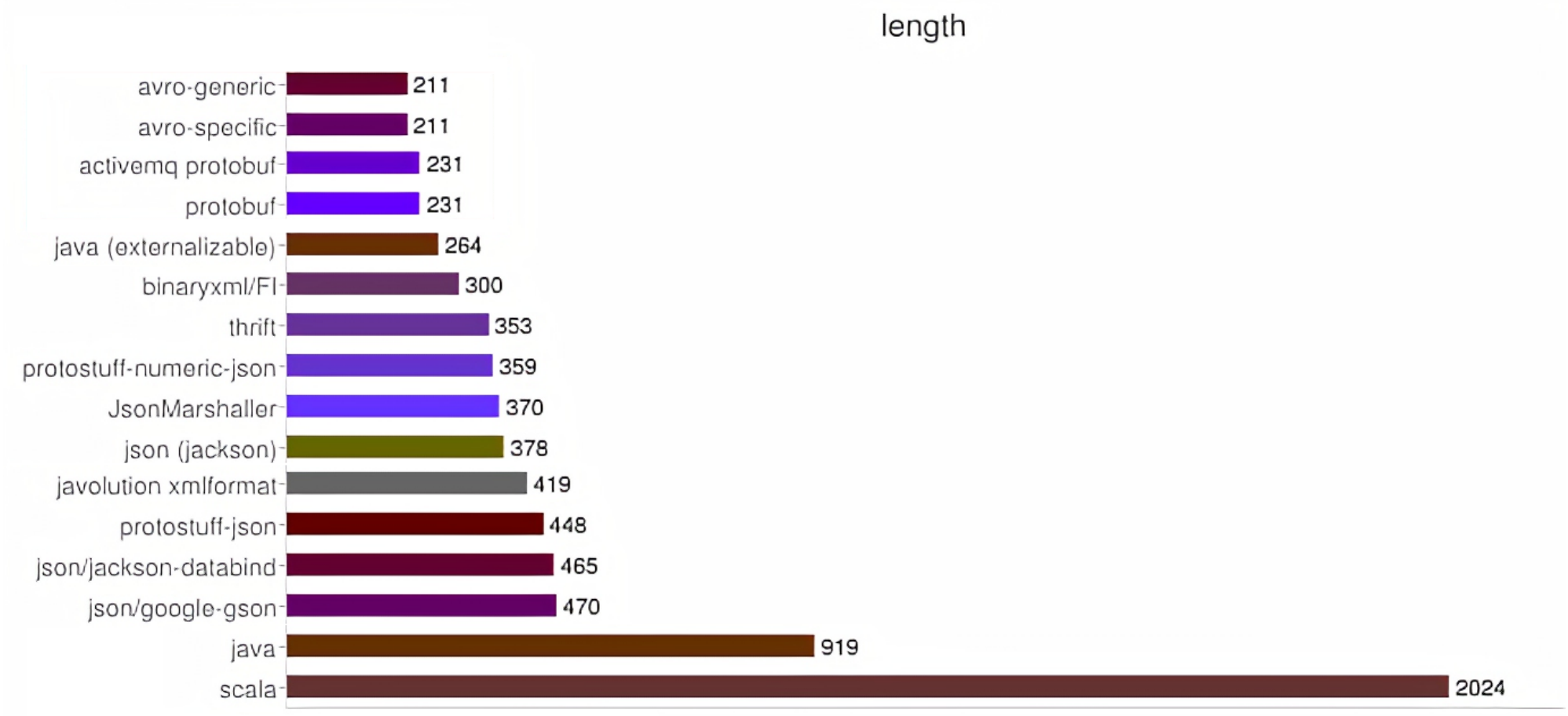
- JSON
  - человеко-читаемый / редактируемый
  - может быть разобран без предварительного знания схемы
  - отличная поддержка браузеров (JavaScript native class description)
  - компактнее, чем XML
- XML
  - человеко-читаемый / редактируемый
  - может быть разобран без предварительного знания схемы
  - стандарт для многих протоколов (SOAP и т.п.)
  - хорошая инструментальная поддержка (XSD, XSLT, SAX, DOM и т.д.)
  - не компактный, большой объем «лишних» описаний
- MessagePack
  - очень компактный (плотная упаковка данных)
  - очень быстрая обработка
  - не предназначен для чтения/редактирования человеком
- Protobuf (Google):
  - очень компактный (плотная упаковка данных)
  - очень быстрая обработка
  - не предназначен для чтения/редактирования человеком
  - встроенная поддержка версий протокола (при изменении протокола, клиенты могут работать со старой версией, пока не обновятся)
  - без знания схемы очень тяжело разобрать (бинарный формат данных, внутренне не однозначен, требуется схема для разбора)

Сравнение времени сериализации

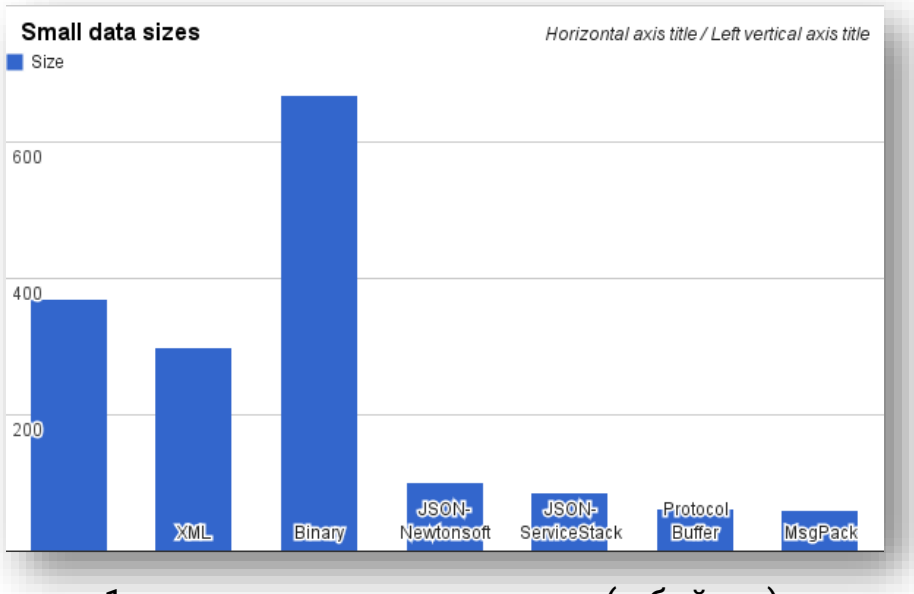


https://blog.softwaremill.com/the-best-serialization-strategy-for-event-sourcing-9321c299632b

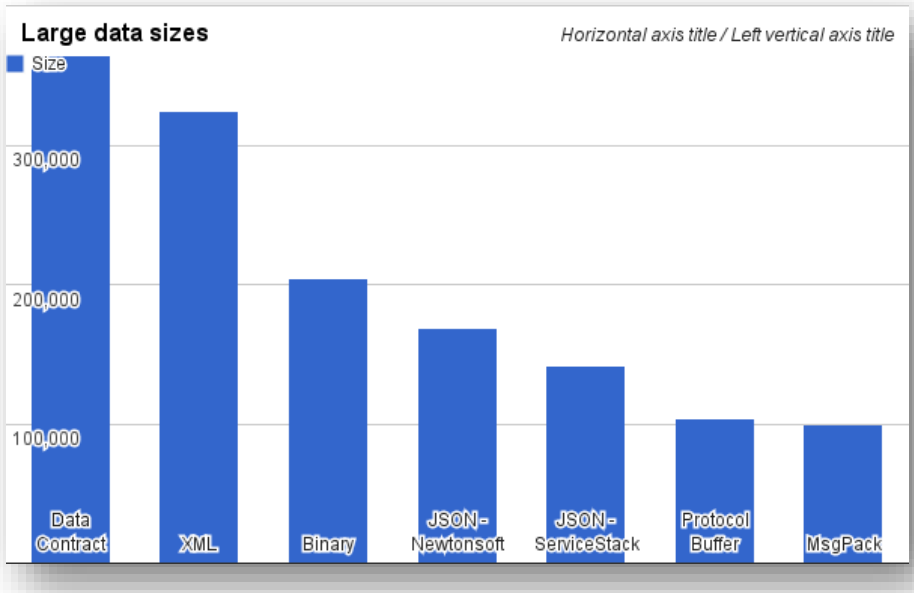
Сравнение размера данных



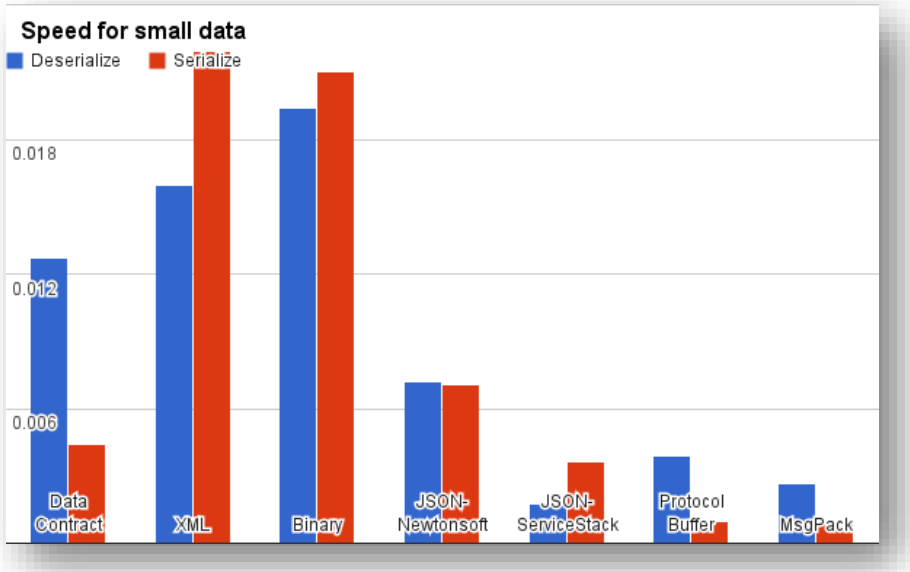
https://blog.softwaremill.com/the-best-serialization-strategy-for-event-sourcing-9321c299632b

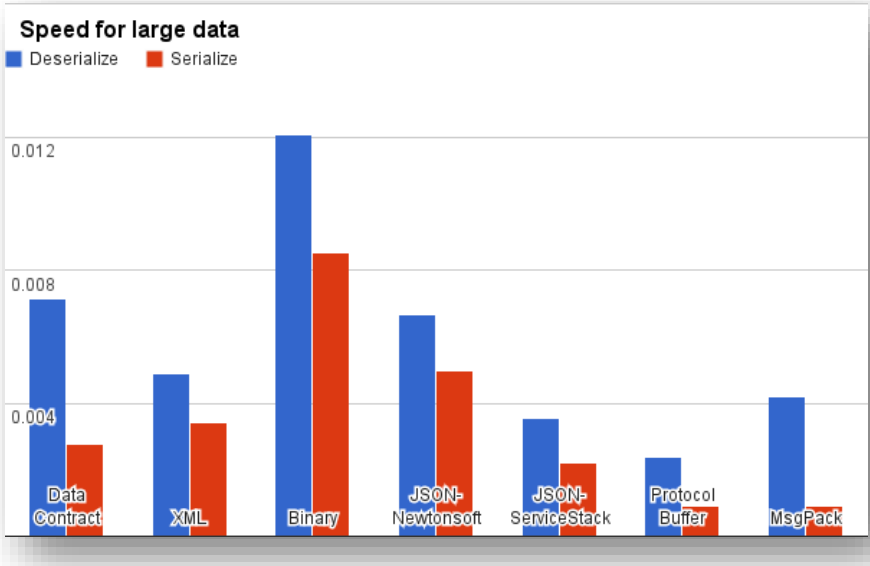


1 запись текстовых данных (в байтах)



1610 записей текстовых данных (в байтах)





https://maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserializer-xmlserializer-binaryformatter-json-newtonsoft-servicestack-text/

Когда использовать какие форматы?

- XML
  - Если система предоставляет публичный API в виде XML Веб-сервиса (изучим их позже)
  - Работаете с «классической» системой, в которой уже используется XML в качестве стандарта обмена данными
  - Требуются стандартные инструменты верификации и трансформации (например, в HTML) (XSD, XSLT, SAX, DOM и т.д.)
- JSON
  - Если система предоставляет публичный API в виде REST-сервиса (изучим их позже)
  - Если клиенты, в большинстве своем, реализуются на JavaScript
  - Более компактный чем XML (в 2-2.5 раза) и самый простой для чтения/редактирования – соответственно, везде, где вы хотели бы использовать XML, подумайте, может быть стоит использовать JSON.
- MessagePack – если требуется высокая скорость и компактность, совместимость с JSON, но не требуется редактирование/чтение человеком
- Protobuf
  - Если для вас прежде всего важен объем и скорость обработки данных
  - Если важна возможность простого обновления протокола
  - Если реализуется внутренний (не публичный) протокол обмена

Спасибо за внимание!

Готов ответить на ваши вопросы!