Сетевое взаимодействие

Вычислительная сеть — это способ организации взаимодействия между двумя и более вычислительными средствами.

Сеть обеспечивает доставку данных от отправителя к получателю.

Как правило, сеть обеспечивает передачу данных между разными компьютерами, но те же технологии могут применяться и в пределах разных встроенных модулей одного компьютера.

Сеть является потенциально ненадежной средой передачи и потому требует особых средств организации и контроля работы.

Сетевое взаимодействие

Данные, передаваемые по сети — это обычные байты, фундаментально не имеющие каких то различий или специального назначения.

Протокол передачи данных — это набор соглашений, позволяющий описать обмен данными и обеспечивающий, в числе прочего, выделение полезных данных из общего информационного потока.

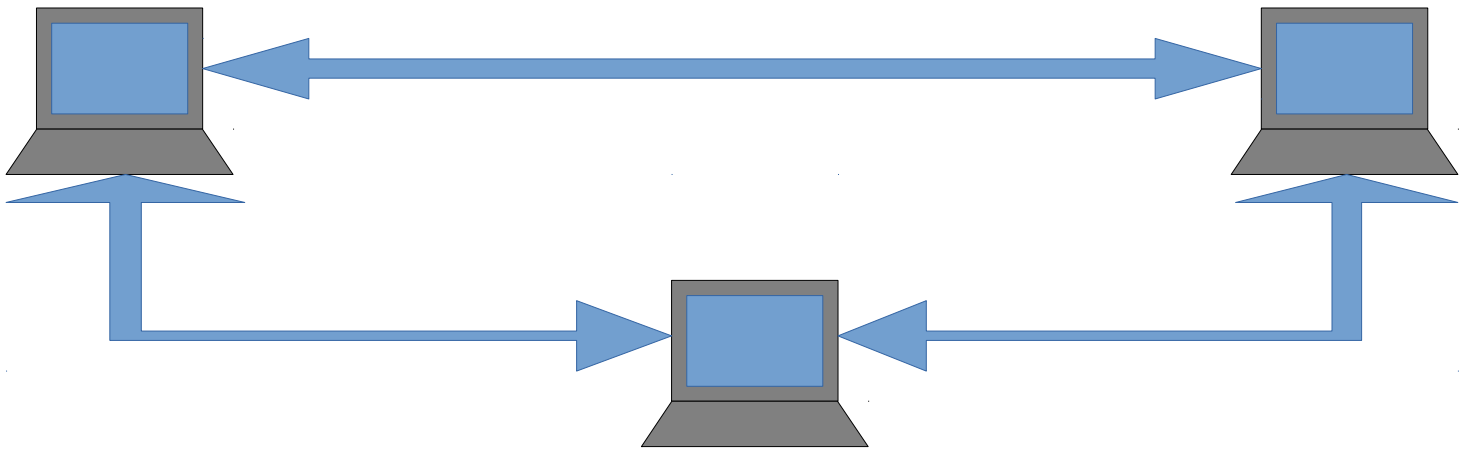
Все участники сетевого взаимодействия обязаны заранее договориться об используемых протоколах и их параметрах.

Коммутация каналов

- Прямое соединение двух компьютеров между собой.
- Не требует промежуточных устройств.
- Требует настройки взаимодействующих устройств.
- Линия передачи может быть использована на полную мощность.

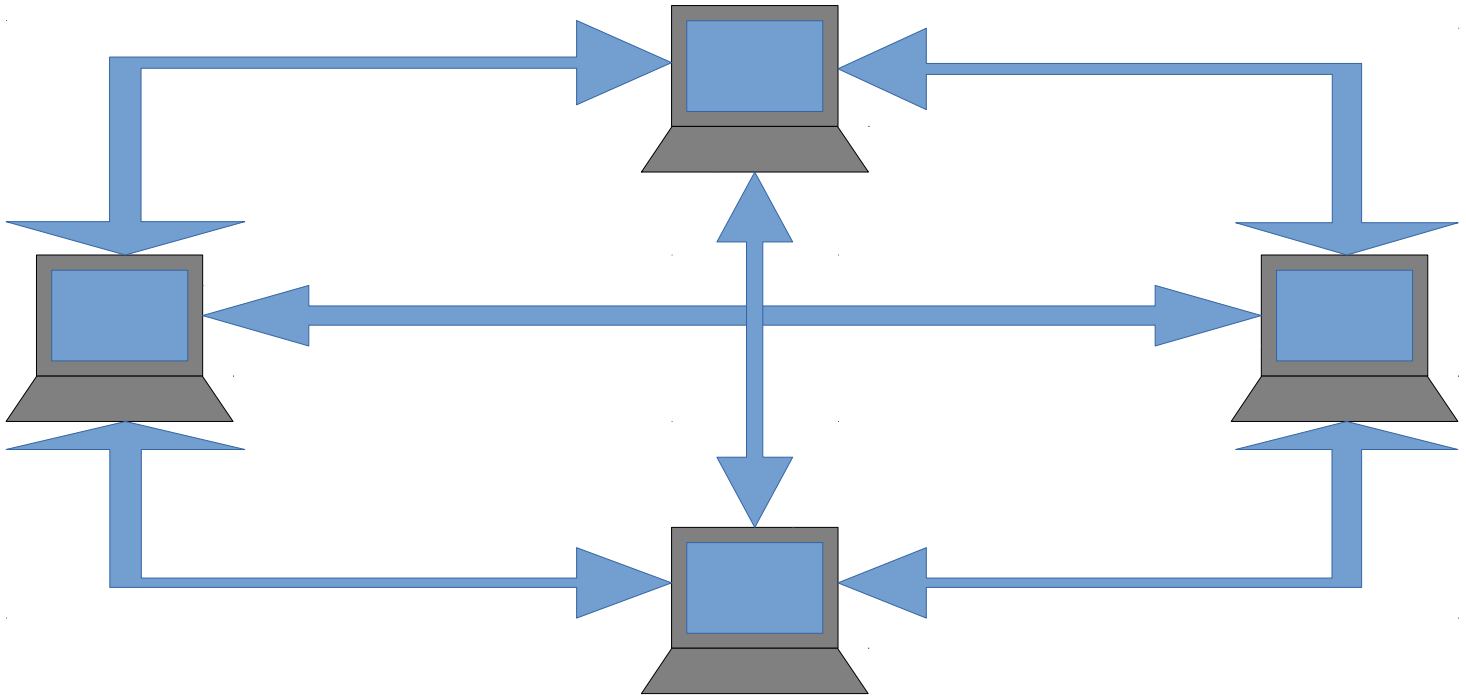
Коммутация каналов

- Прямое соединение двух компьютеров между собой.
- Не требует промежуточных устройств.
- Требует настройки взаимодействующих устройств.
- Линия передачи может быть использована на полную мощность.



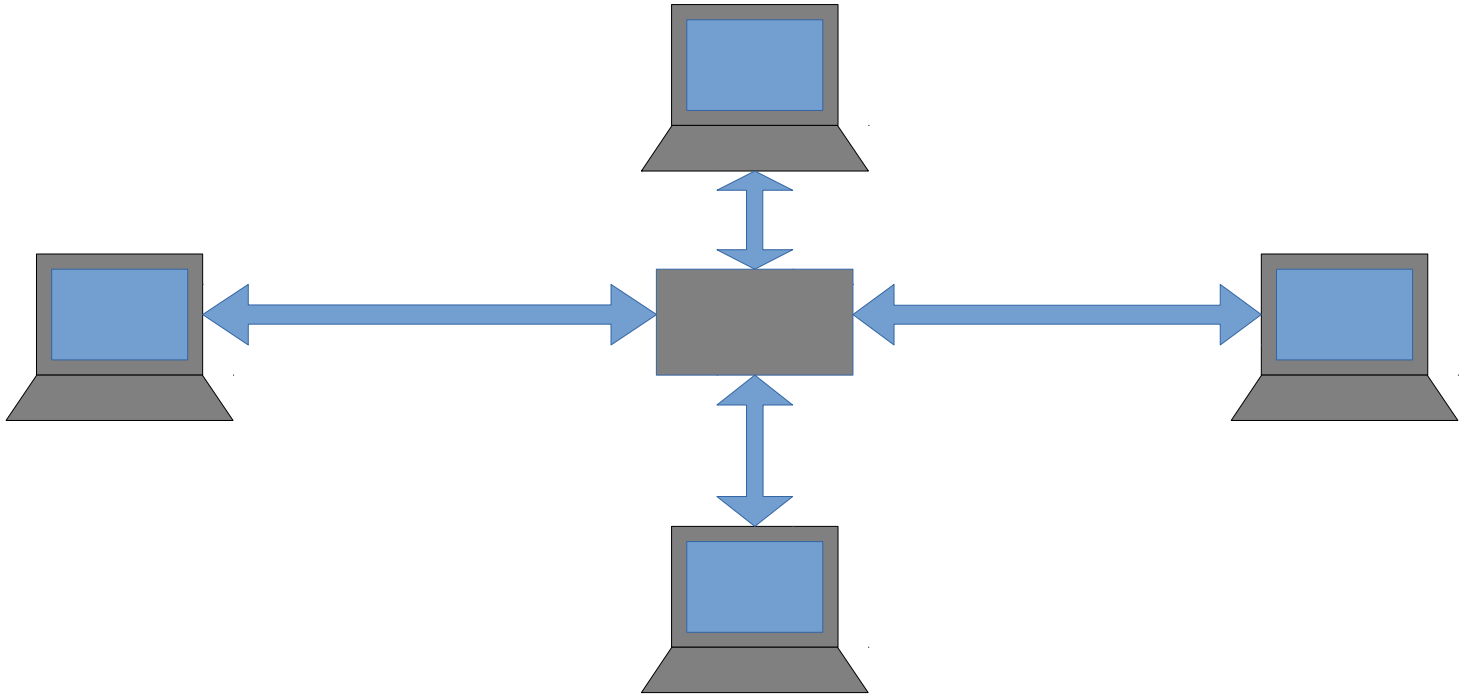
Коммутация каналов

- Прямое соединение двух компьютеров между собой.
- Не требует промежуточных устройств.
- Требует настройки взаимодействующих устройств.
- Линия передачи может быть использована на полную мощность.



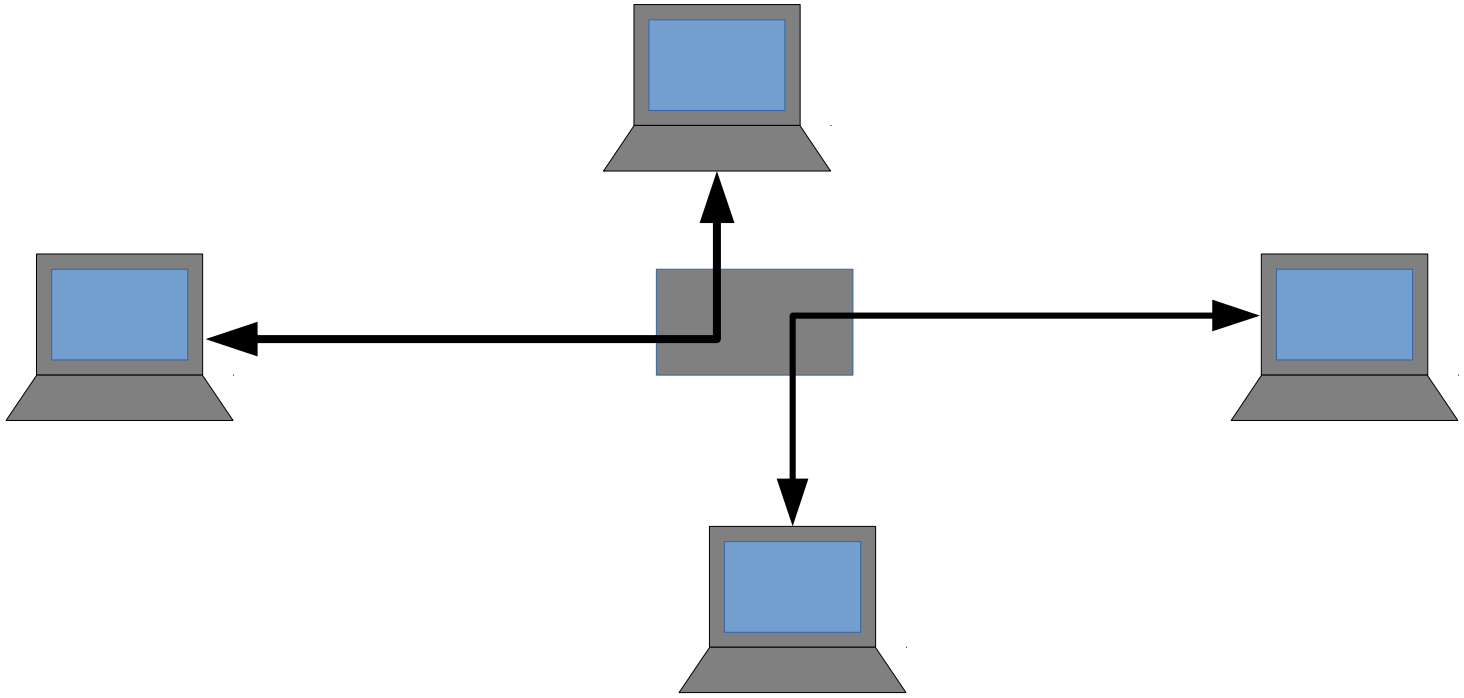
Коммутация каналов

- Прямое соединение двух компьютеров между собой.
- Не требует промежуточных устройств.
- Требует настройки взаимодействующих устройств.
- Линия передачи может быть использована на полную мощность.
- Требует наличия промежуточного узла — коммутатора.



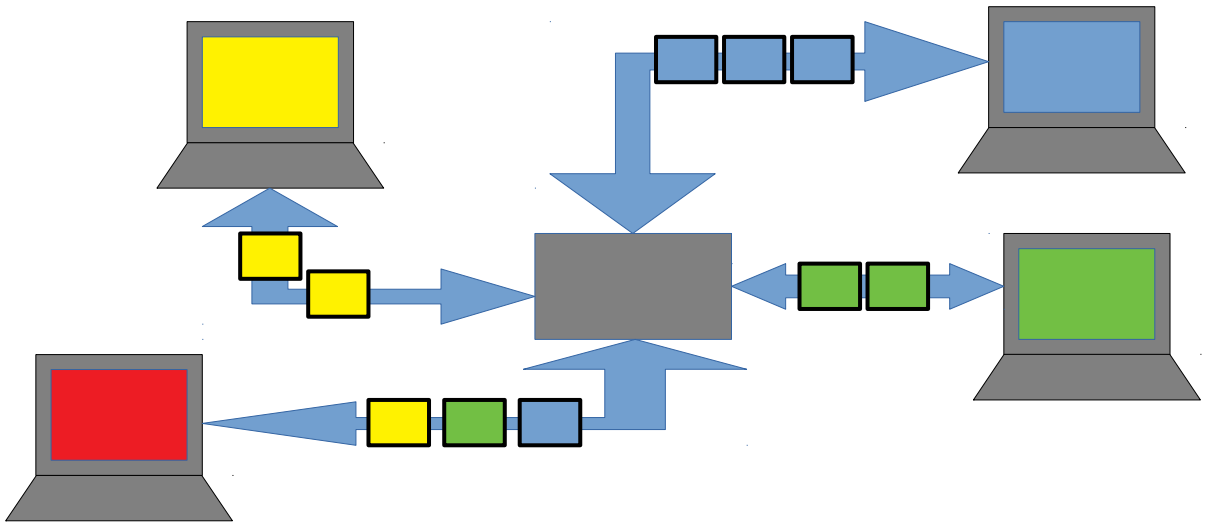
Коммутация каналов

- Прямое соединение двух компьютеров между собой.
- Не требует промежуточных устройств.
- Требует настройки взаимодействующих устройств.
- Линия передачи может быть использована на полную мощность.
- Требует наличия промежуточного узла — коммутатора.



Коммутация пакетов

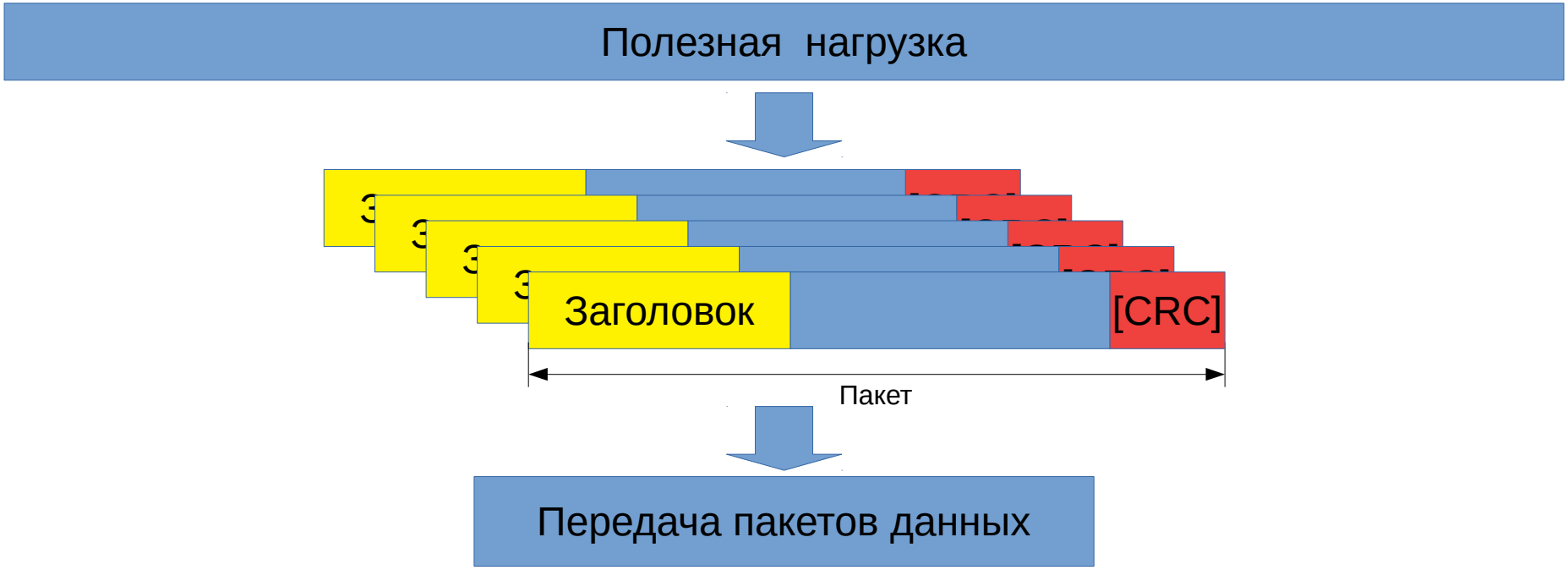
- **Пакет** — это небольшая порция от общего объема передаваемых данных. Каждый пакет содержит информацию об отправителе и получателе.
- Все узлы сети соединены друг с другом в любой момент времени. Возможна одновременная передача данных разным узлам путем чередования пакетов.
- Повышается сложность коммутационного оборудования: требуется анализ проходящих данных.



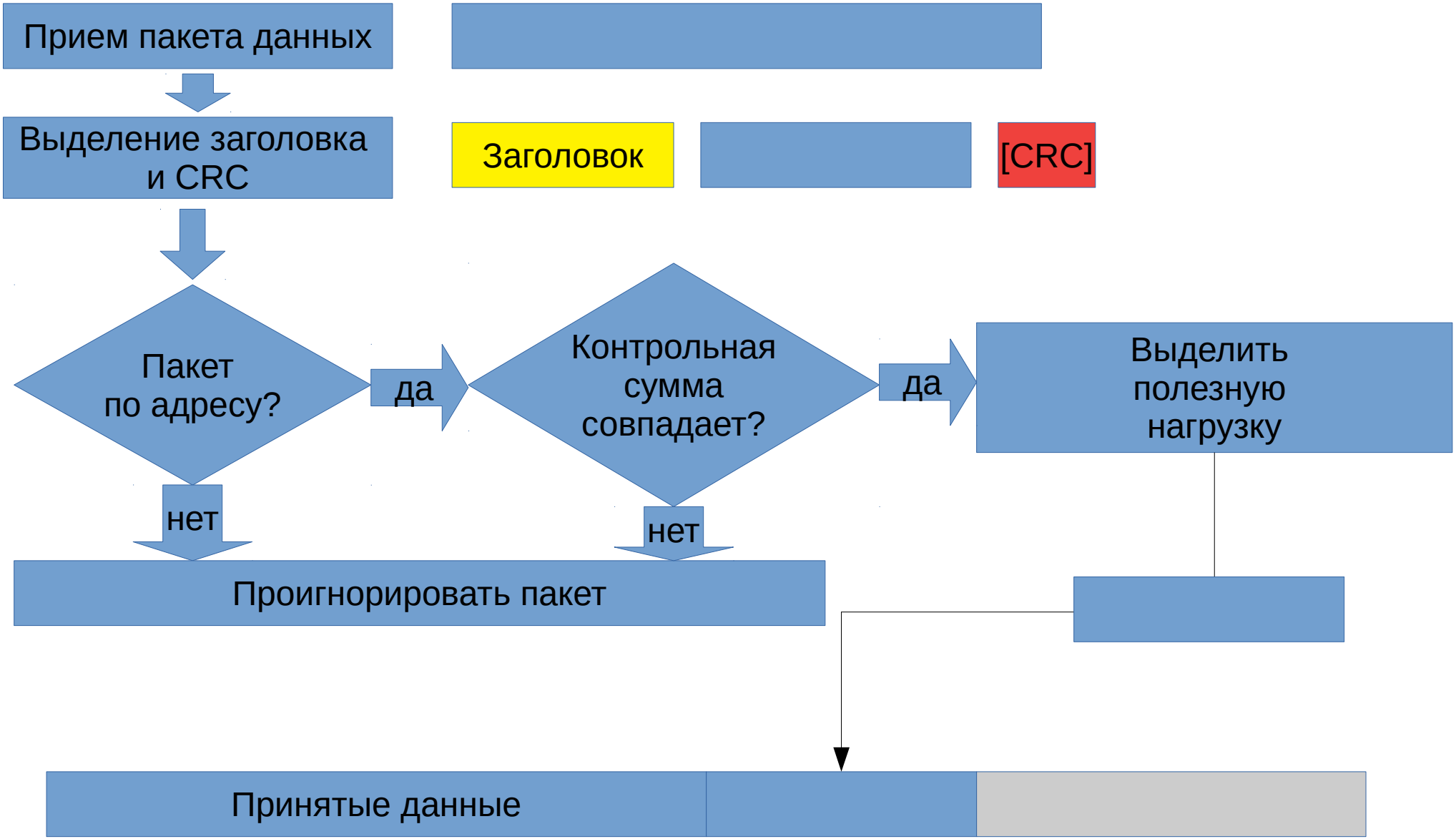
Формирование пакета данных

- Общий объем передаваемой информации нарезается на отдельные блоки конечного объема, определенного протоколом взаимодействия.
- К каждому блоку добавляется заголовок, содержащий информацию об отправителе и получателе. Блок исходных данных в пакете носит название **полезная нагрузка** (payload).
- Также к пакету может добавляться "хвост", содержащий контрольную сумму (CRC).
- Принимающая сторона отделяет заголовок и анализирует его. Если пакет пришел по адресу, полезная нагрузка выделяется и добавляется к полезной нагрузке, принятой ранее.
- При передаче без ошибок, приемная сторона собирает у себя копию передаваемых данных. В случае ошибок, может быть запрошена повторная отправка потерянных пакетов.

Передача данных в виде пакетов



Прием пакета данных



Классификация протоколов

На сегодняшний день существует множество сетевых протоколов, решающих разные задачи.

Наиболее полную классификацию протоколов предлагает модель OSI (Open System Interface). Она разделяет протоколы на 7 уровней, в зависимости от назначения.

Полезные данные при передаче через сеть проходят все 7 уровней модели, на каждом из которых они преобразуются для решения отдельной задачи.

На приемной стороне данные проходят те же уровни в обратном порядке.

Существует упрощенная модель: DOD. Она содержит 4 уровня и получается из модели OSI путем объединения первых двух и последних трех уровней.

Классификация протоколов

Прикладной уровень

- Формирует данные, которые необходимо передать.
- Прикладное ПО работает именно на этом уровне.
- Примеры протоколов:
  - HTTP (Hyper-Text Transfer Protocol)
  - SMTP (Simple Mail Transfer Protocol)
  - POP (Post Office Protocol)
  - FTP (File Transfer Protocol)
  - ...

Представительский уровень

- Используется, когда требуется сжатие или шифрование данных.
- Планировался в том числе для преобразования кодировки данных.
- В большинстве случаев отсутствует.

Сеансовый уровень

- Обеспечивает установку логического соединения поверх других протоколов.
- Как правило, добавляет шифрование данных.
- Может обеспечить создание виртуального сетевого устройства.
- Пример протоколов:
  - PPTP (Point-to-Point Tunneling Protocol)
  - L2TP (Level 2 Tunneling Protocol)
  - PAP (Password Authentication Protocol)
  - NetBIOS

Транспортный уровень

- Обеспечивает доставку данных в полном объеме и в правильном порядке.
- Начиная с транспортного уровня, данные разбиваются на отдельные куски — сегменты.
- Пример протоколов:
  - TCP (Transmission Control Protocol). Стандарт де-факто при использовании в локальных сетях и интернете.
  - UDP (User Datagram Protocol). Обеспечивает полную доставку или недоставку одного сегмента — датаграммы .

Сетевой уровень

- Обеспечивает адресацию пакетов.
- Вводит понятие сетевого адреса: программно изменяемого идентификатора устройства.
- Сегменты данных разбиваются на отдельные пакеты.
- Пример протоколов:
  - IP (Internet Protocol). Стандарт де-факто при использовании в локальных сетях и интернете.
  - RIP (Route Information Protocol)
  - OSPF (Open Shortest Path First)

Канальный уровень

- Обеспечивает преобразование пакетов в двоичные данные и обратно.
- Работает на «прямой» между отдельными устройствами.
- Вводит понятие физического адреса.
- Тесно переплетен с физическим уровнем.
- Пример протоколов: Ethernet, FDDI (Fiber Distributed Data Interface), Wireless LAN (IEEE 802.11), PPP (Point-to-Point Protocol), PPPoE (Point-to-Point Protocol over Ethernet)

Физический уровень

- Обеспечивает кодирование двоичных данных в виде физических сигналов.
- Определяет набор стандартов как для программного, так и для аппаратного обеспечения.
- Тесно переплетен с канальным уровнем.
- Пример протоколов:
  - Семейство IEEE 802 (802.3 — Ethernet, 802.11 — WiFi, 802.15 — Bluetooth, 802.16 — WiMAX);
  - GSM;
  - RS-232, RS-485.

Стек протоколов TCP/IP

- Наиболее распространенная реализация модели OSI.
- Является отраслевым стандартом для передачи данных по локальным сетям и через интернет.
- Обеспечивает однозначную адресацию и целостность доставки данных.
- Поддерживается сетевым оборудованием.
- Может бесшовно работать поверх большинства известных протоколов физического и канального уровня.
- Можно использовать протокол прикладного уровня напрямую поверх TCP/IP.

Протокол TCP

- Transmission Control Protocol
- Вводит 32-битные параметры: номер последовательности и номер подтверждения. По ним можно однозначно установить порядок следования сегментов и определить потерянные сегменты.
- Вводит 16-битные параметры: порт отправителя и порт получателя. Порт позволяет идентифицировать конкретного получателя в пределах одного компьютера.
- Ориентирован на установку соединения. Предусматривает служебные пакеты без полезных данных.
- Контроль целостности происходит при помощи контрольной суммы. На приеме сравнивается контрольная сумма в заголовке сегмента и фактически рассчитанная для принятых данных.

Протокол IP

- Internet Protocol
- Существует параллельно в двух версиях: IPv4 и IPv6.
- Вводит понятие сетевого адреса или IP-адреса. Разные версии протокола отличаются форматом адреса.
- Заголовок IP включает в себя адреса отправителя и получателя.
- Определяет TTL (Time To Live) — специальную переменную в заголовке пакета, которая уменьшается на 1 при каждом проходе через промежуточный узел. При уменьшении TTL до нуля пакет считается безнадежно потерянным.

Адреса IPv4

Адрес IPv4 представляется как 32-битное целое число, либо как четыре 8-битных числа.

В строковом представлении они записываются разделенные точкой.

```
192.168.0.1
```

IP-адрес назначается каждому сетевому интерфейсу. Адрес может быть настроен вручную, либо получен автоматически от специального узла — DHCP-сервера.

В пределах одной сети IP-адреса не должны повторяться.

Адреса IPv4

При адресации в IPv4 старшие биты адреса отводятся на адрес подсети, а младшие — на адрес конкретного устройства.

```
192.168.0.91    - Адрес устройства
192.168.0.0     - Адрес подсети устройства
```

Любое сетевое устройство содержит таблицу маршрутизации, позволяющую по адресу подсети определить направление дальнейшей отправки пакета. Для большинства бытовых устройств существует шлюз «по умолчанию».

Также, адрес 0.0.0.0 зарезервирован и обозначает все адреса текущего устройства.

Адреса IPv4

Один из первых подходов к адресации — использование класса подсети. Класс определяется количеством единичных бит в начале адреса.

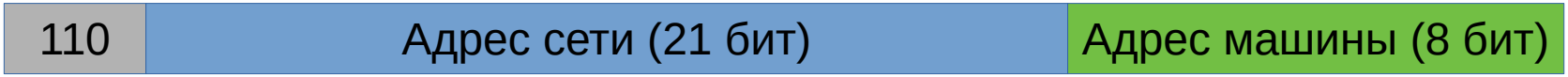
- Класс A

  126 подсетей (все кроме 0 и 127) по 16 млн. адресов.

- Класс B

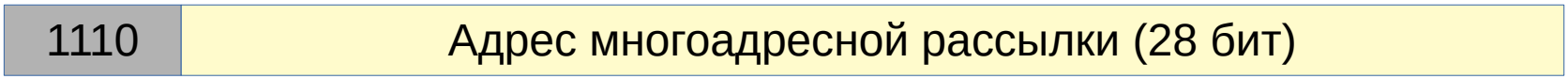
  Порядка 16000 сетей по 65534 адресов.

- Класс C



  Свыше 2 млн сетей по 254 адреса.

- Класс D



  Адреса много адресной рассылки. Специальное назначение.

- Класс E

  Зарезервирован. Эти адреса не использовались никогда.

Адреса IPv4

Маска подсети позволяет обойти ограничения классовой адресации. Маска записывается в той же форме, что и IP-адрес, и всегда содержит некоторое количество идущих подряд единиц, за которыми идут нули. Применение побитового «И» между IP-адресом устройства и маской подсети даст адрес сети устройства.

```
IP:    192.168.1.117  11000000 10100000 00000001 01110101
Маска: 255.255.254.0  11111111 11111111 11111110 00000000
Сеть:  192.168.0.0    11000000 10100000 00000000 00000000
```

Часто применяют сокращенную форму записи маски. Маска записывается как число единичных бит после символа «/».

В примере выше, запись 255.255.254.0 эквивалентна записи /23.

Короткая форма записи: 192.168.1.117/23

Емкость сети равна 2^{32-M}, где M — число бит маски.

Специальные адреса IPv4

- 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 — адреса для использования в частных сетях. Недоступны для использования в сети интернет.
- 127.0.0.0/8 — интерфейс обратной связи.
- 169.254.0.0/16 — канальные адреса. Присваиваются операционной системой, если устройству не назначен никакой другой адрес.
- 0.0.0.0/8 — «своя» подсеть, то есть текущая подсеть, в которой находится устройство.
- 240.0.0.0/4 — подсеть класса E, зарезервирована и запрещена к использованию.
- «Последний» адрес сети, когда все биты адреса машины равны 1, используется для широковещательных пакетов в пределах подсети. Например, адрес 10.0.1.255 в сети 10.0.1.0/24.

Ограничение объема адресов

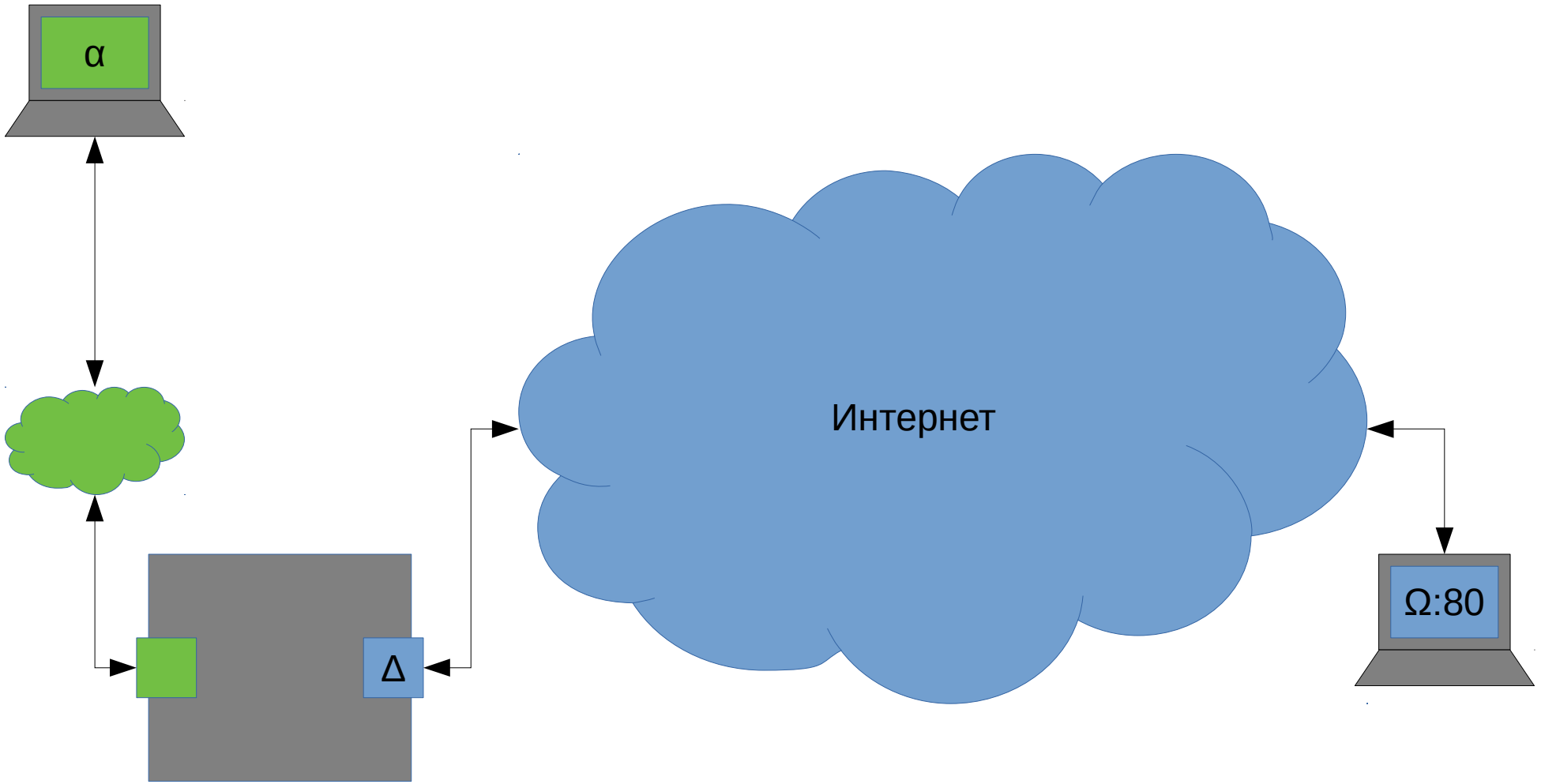
Емкость адресного пространства IPv4 — около 4 млрд. адресов. С учетом зарезервированных адресов — около 3.7 млрд. К концу 2019 года были распределены все свободные адреса IPv4.

Один из вариантов временного решения — разделение на общие («белые») и частные («серые») адреса, а также введение механизма NAT (Network Address Translation), позволяющего преобразовывать адреса на лету.

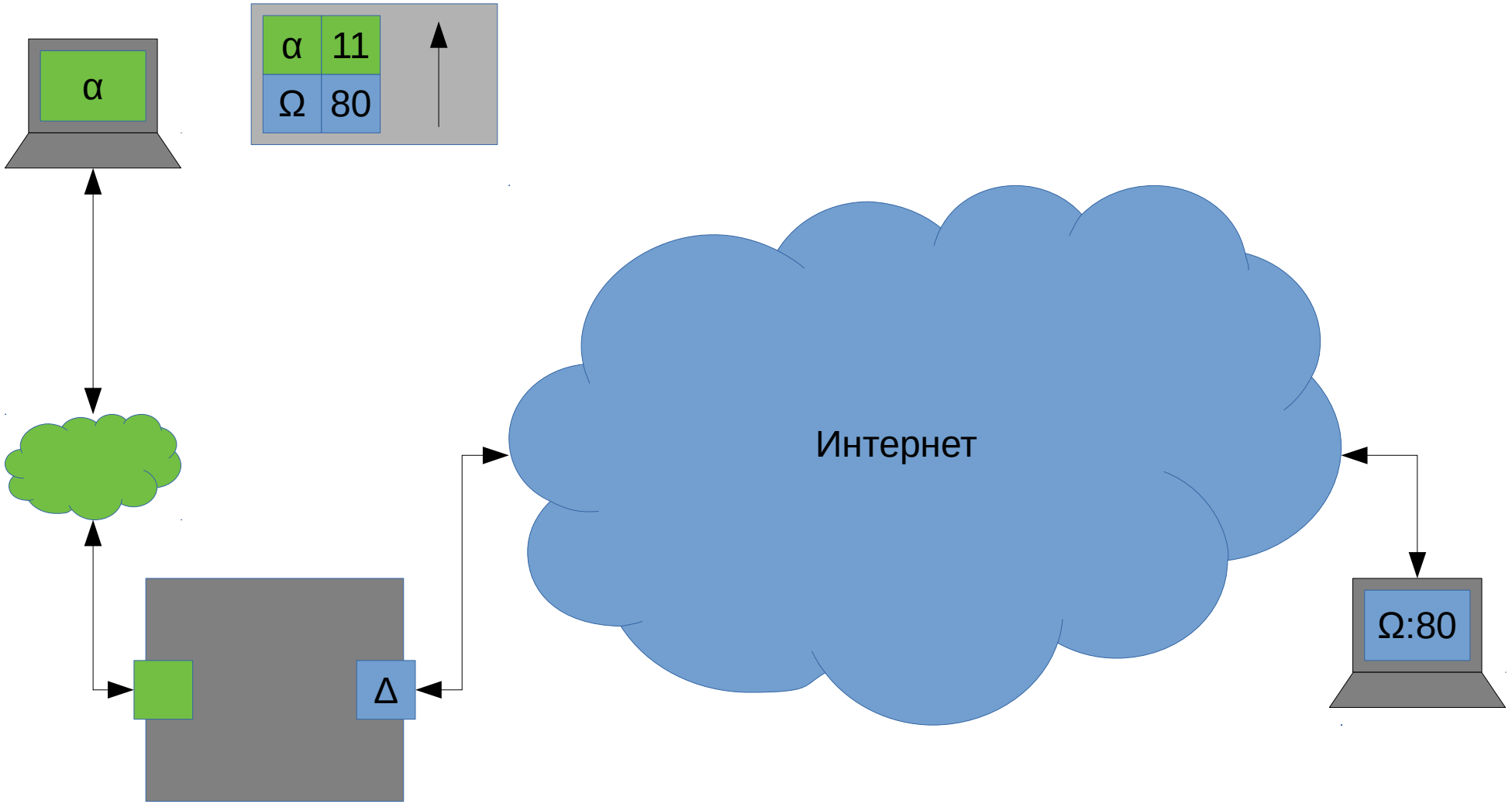
Устройство с NAT (например, маршрутизатор) подменяет адрес и порт источника в исходящем пакете и запоминает у себя в памяти подмену. Устройство производит обратную подмену в ответных пакетах. Для конечных устройств этот процесс прозрачен.

Частные адреса в разных подсетях изолированы друг от друга, поэтому их количество фактически не ограничено.

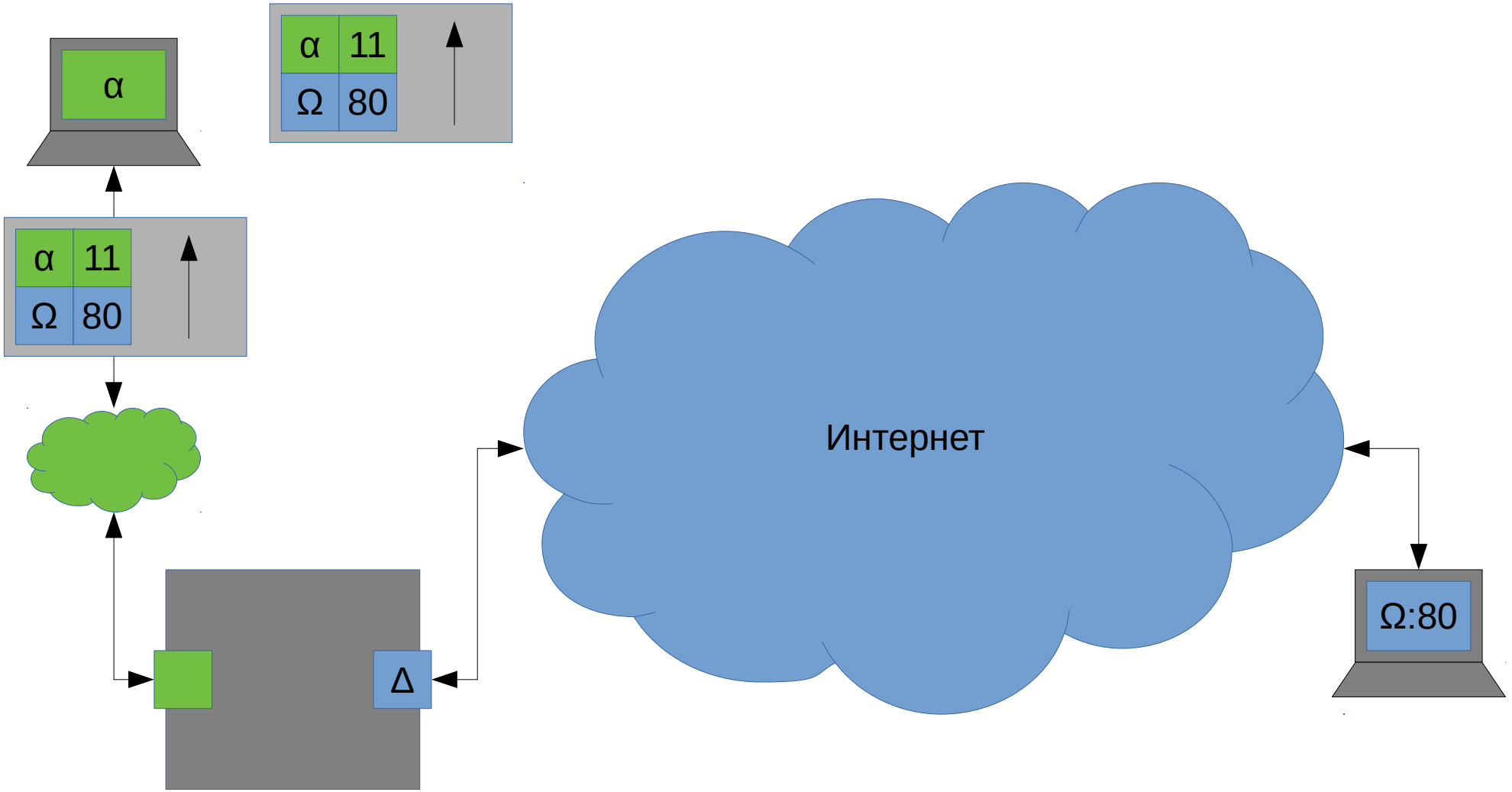
NAT



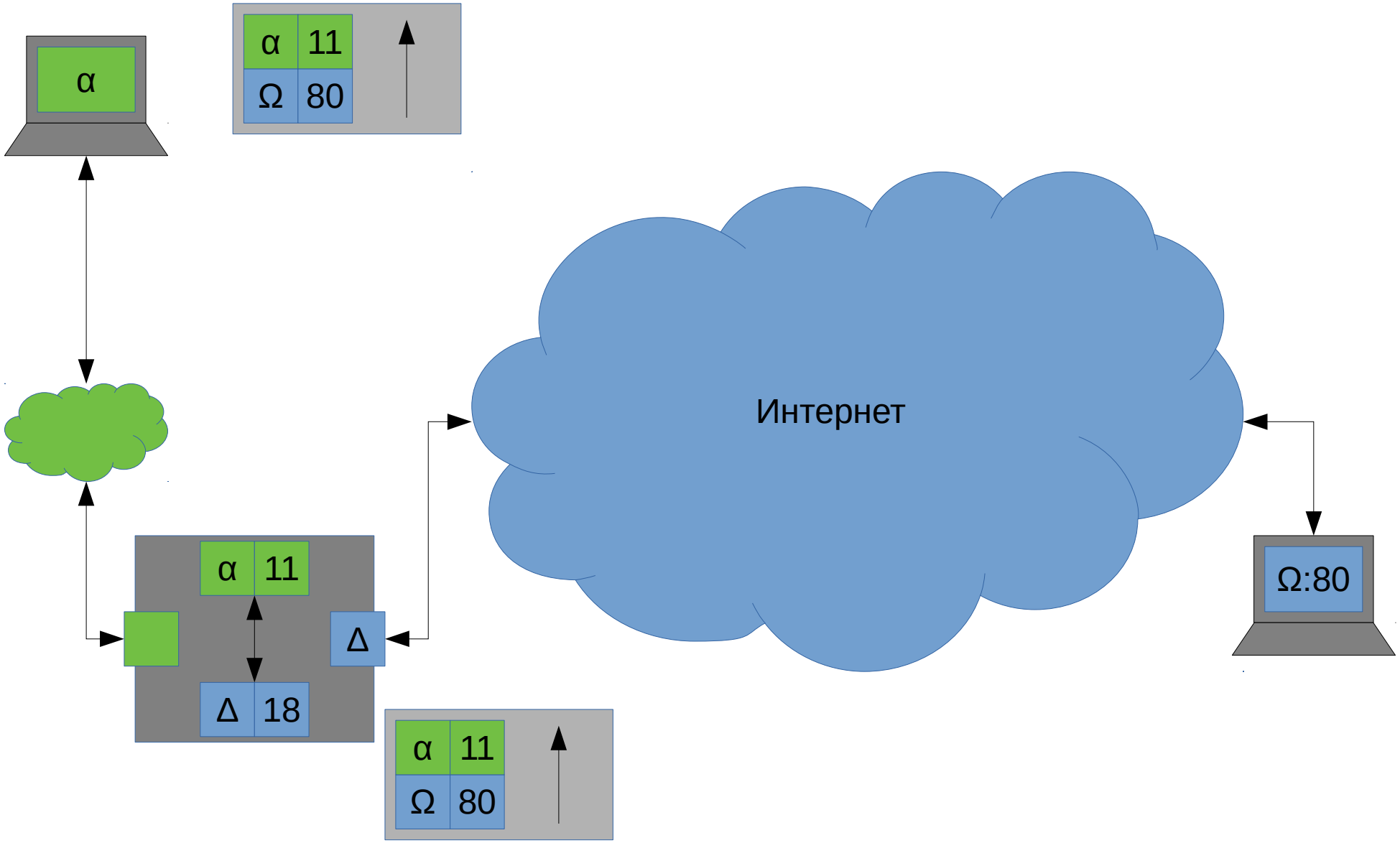
NAT



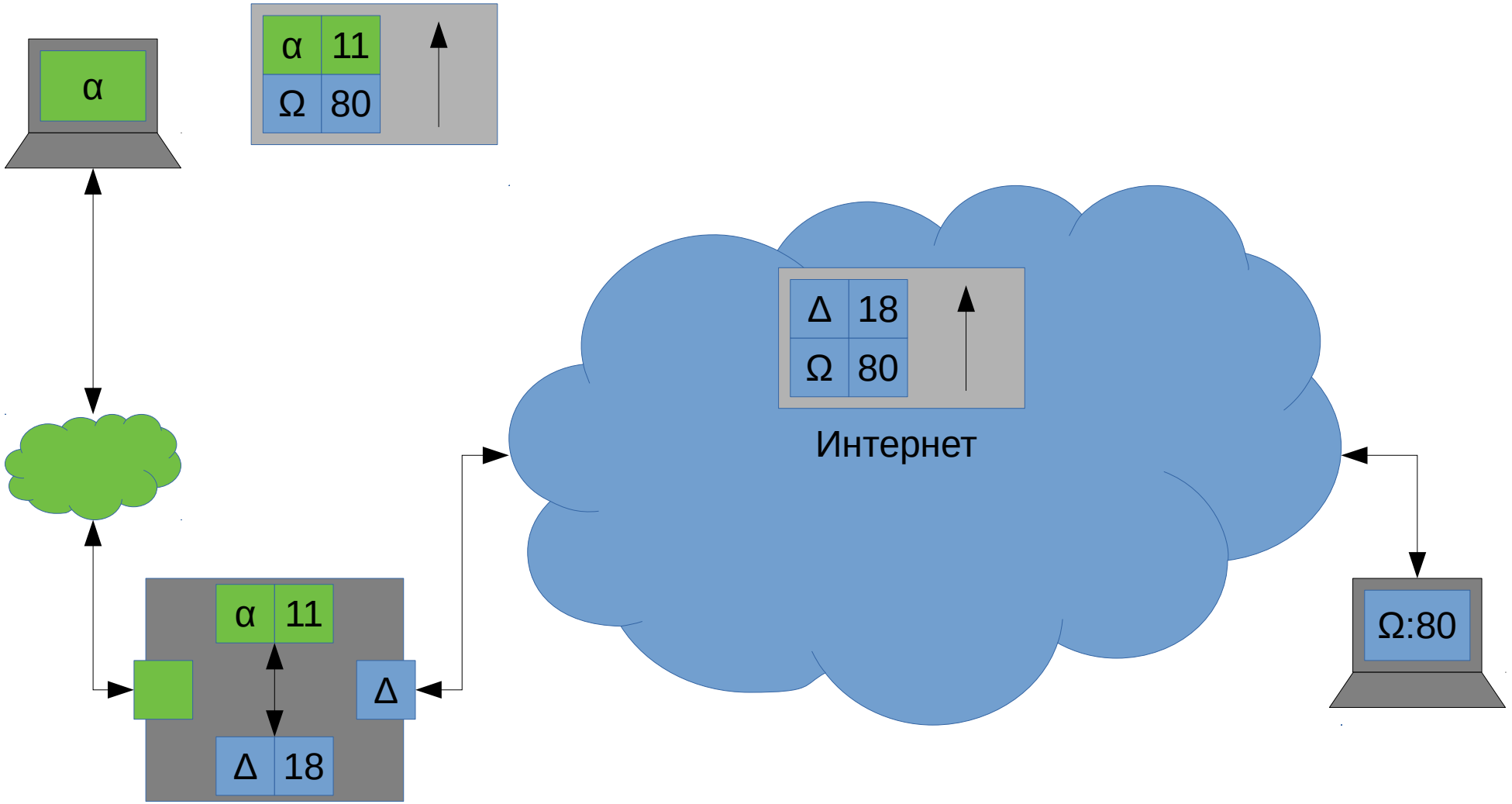
NAT



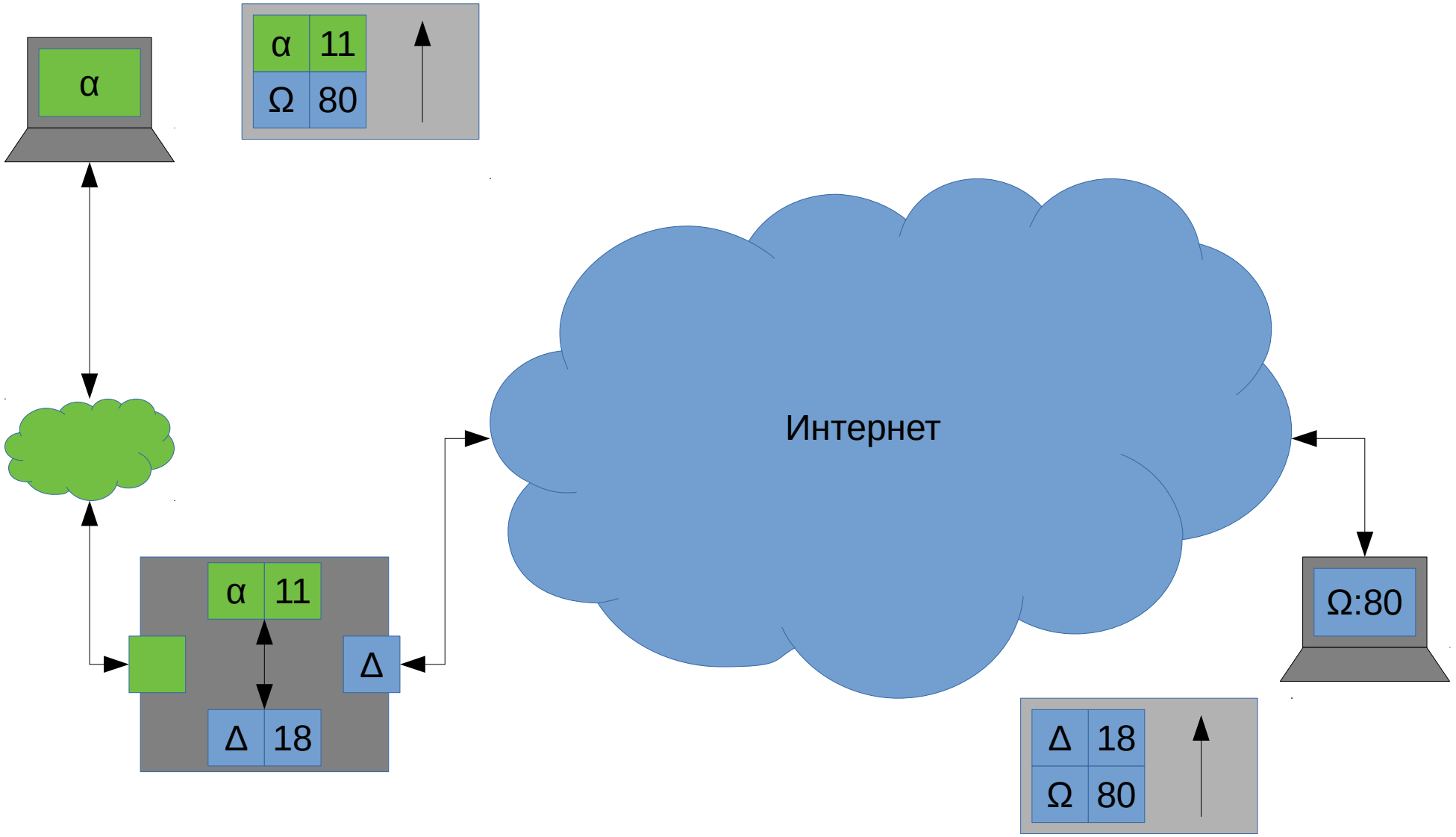
NAT



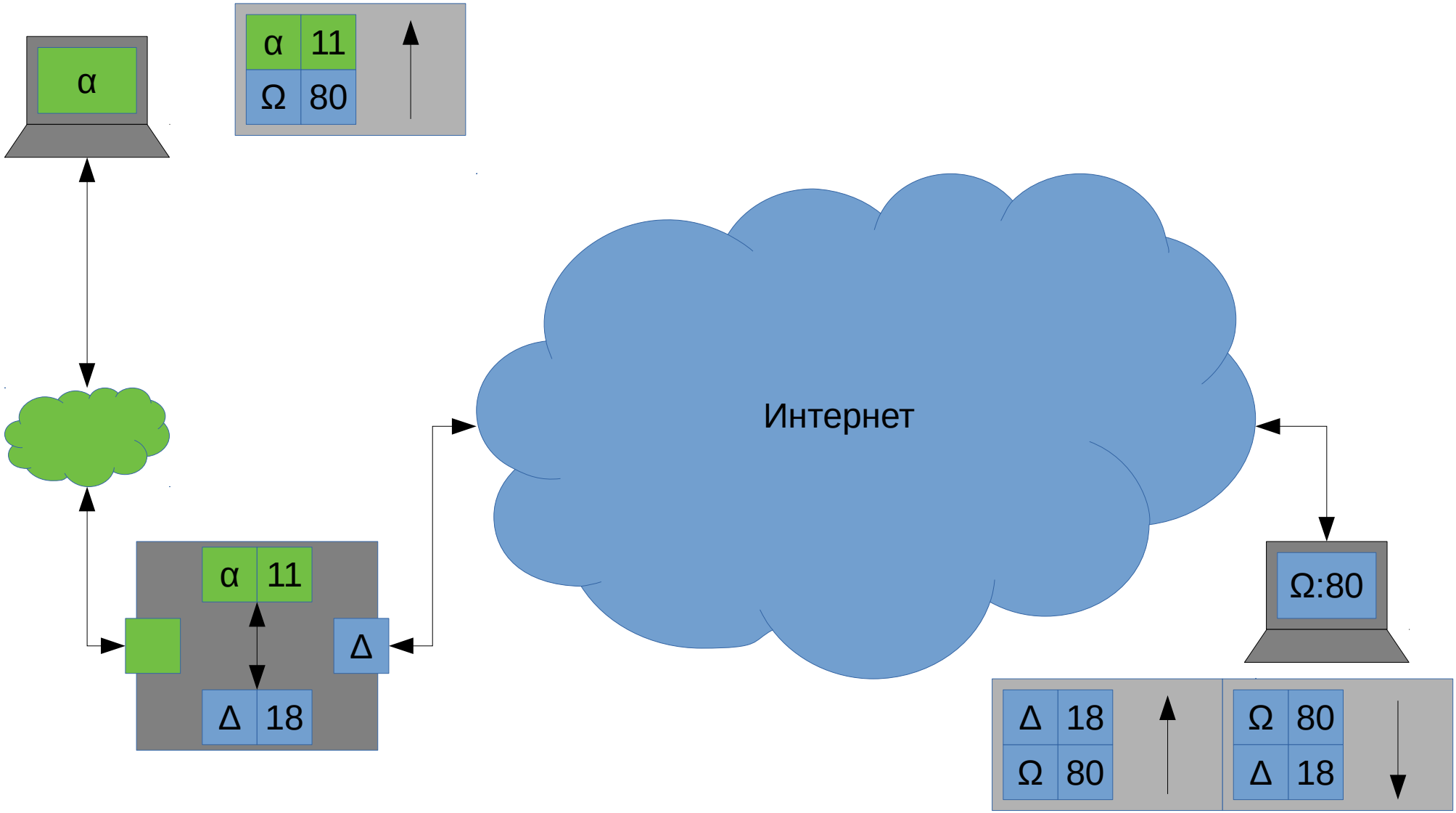
NAT



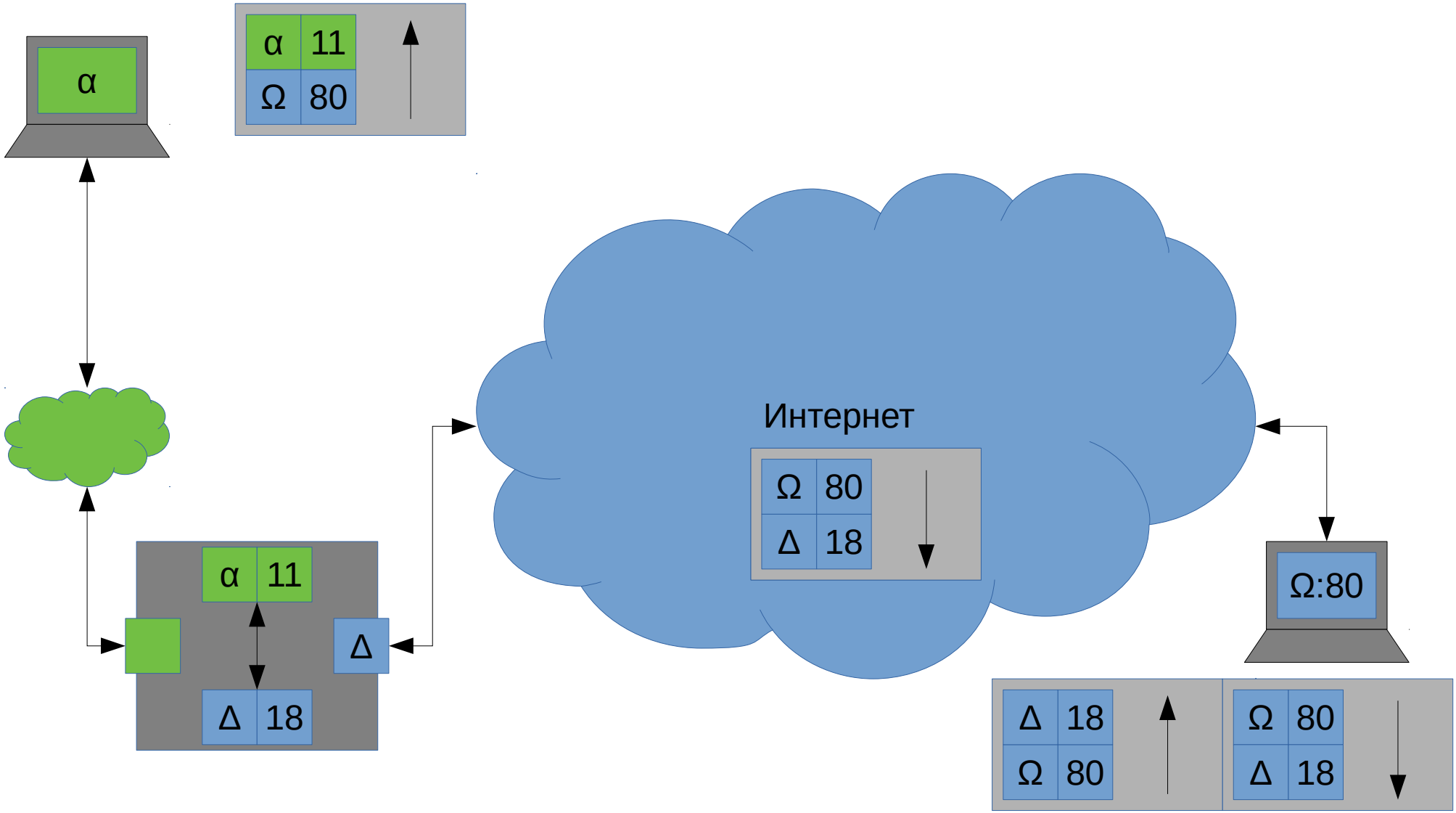
NAT



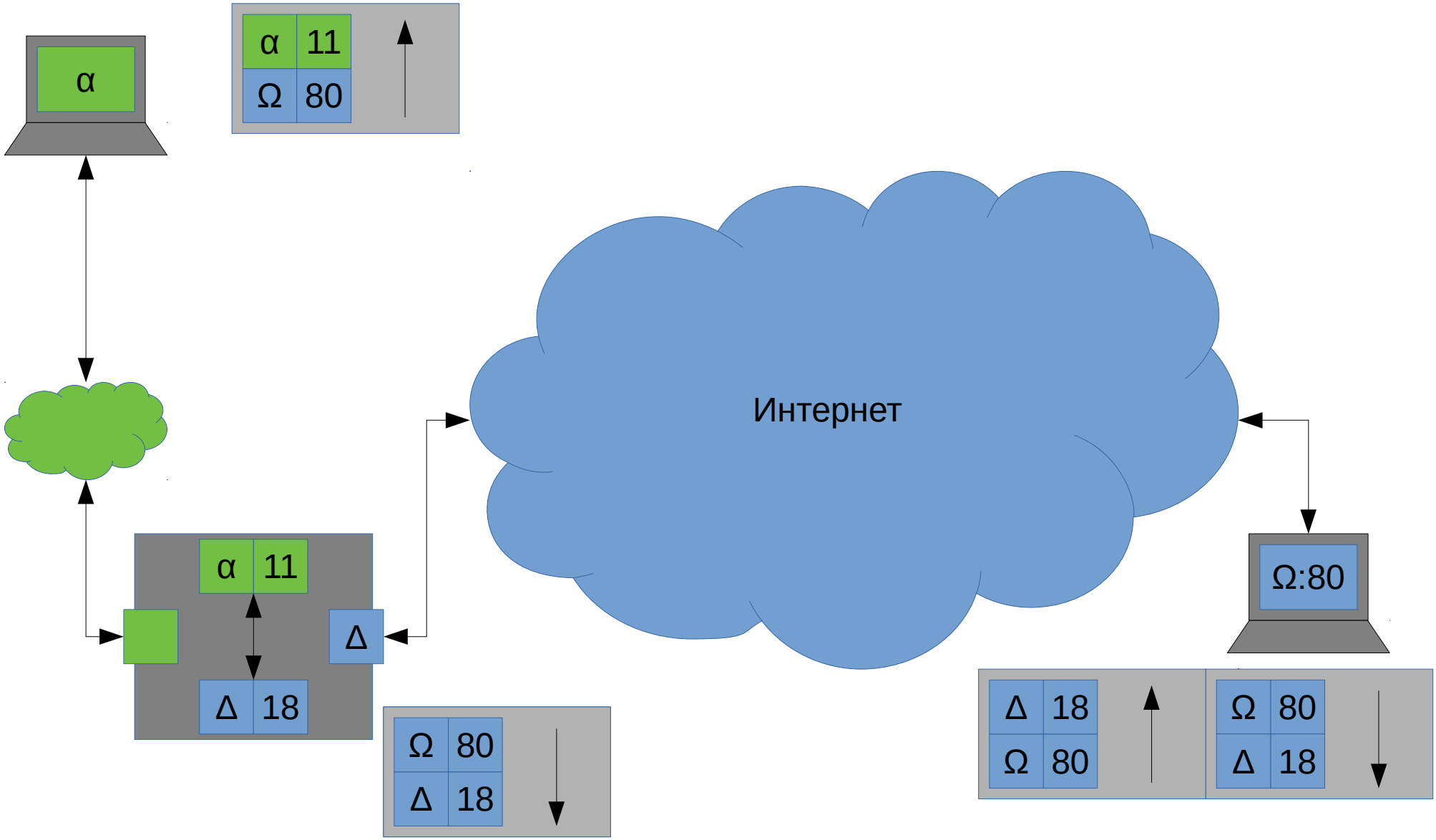
NAT



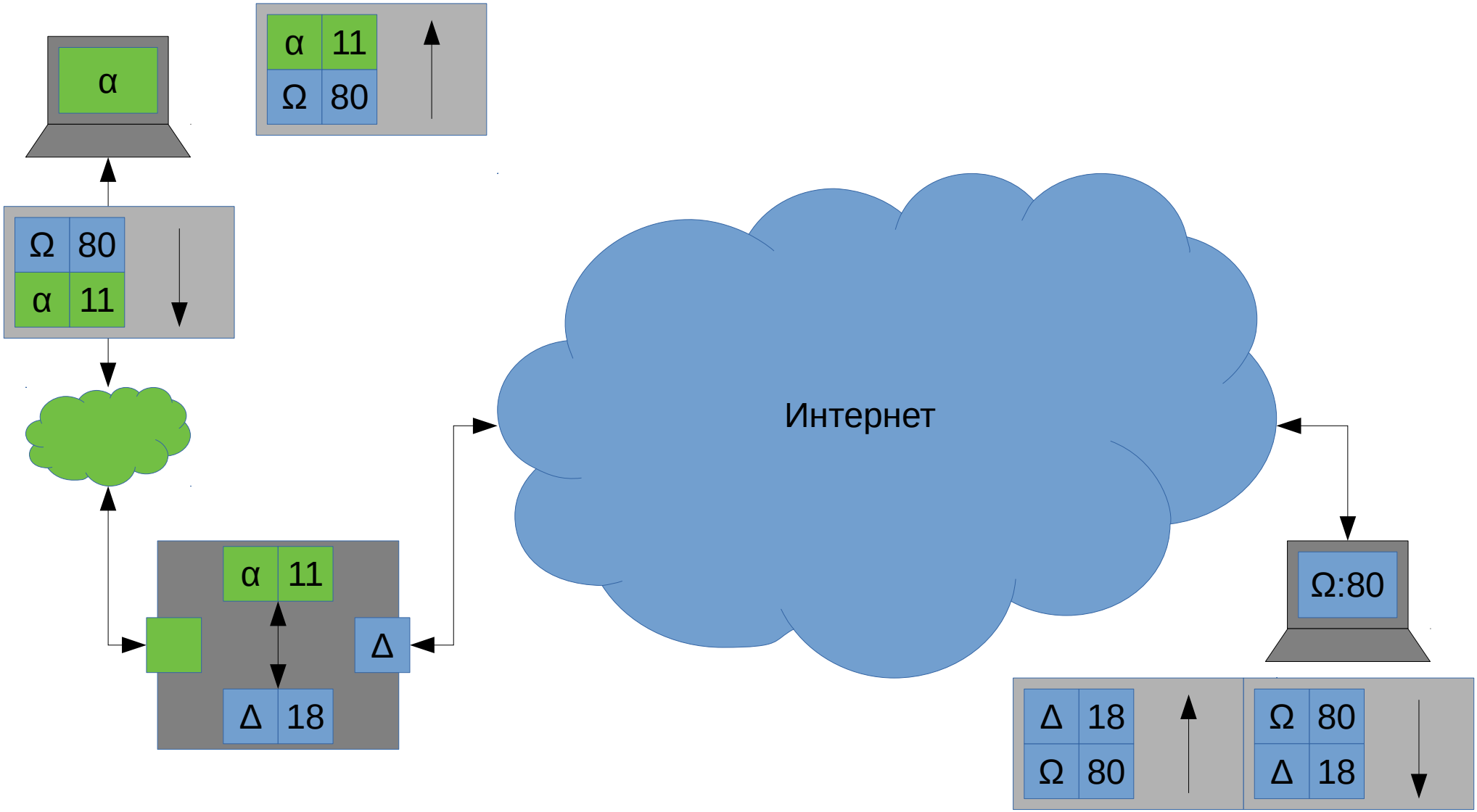
NAT



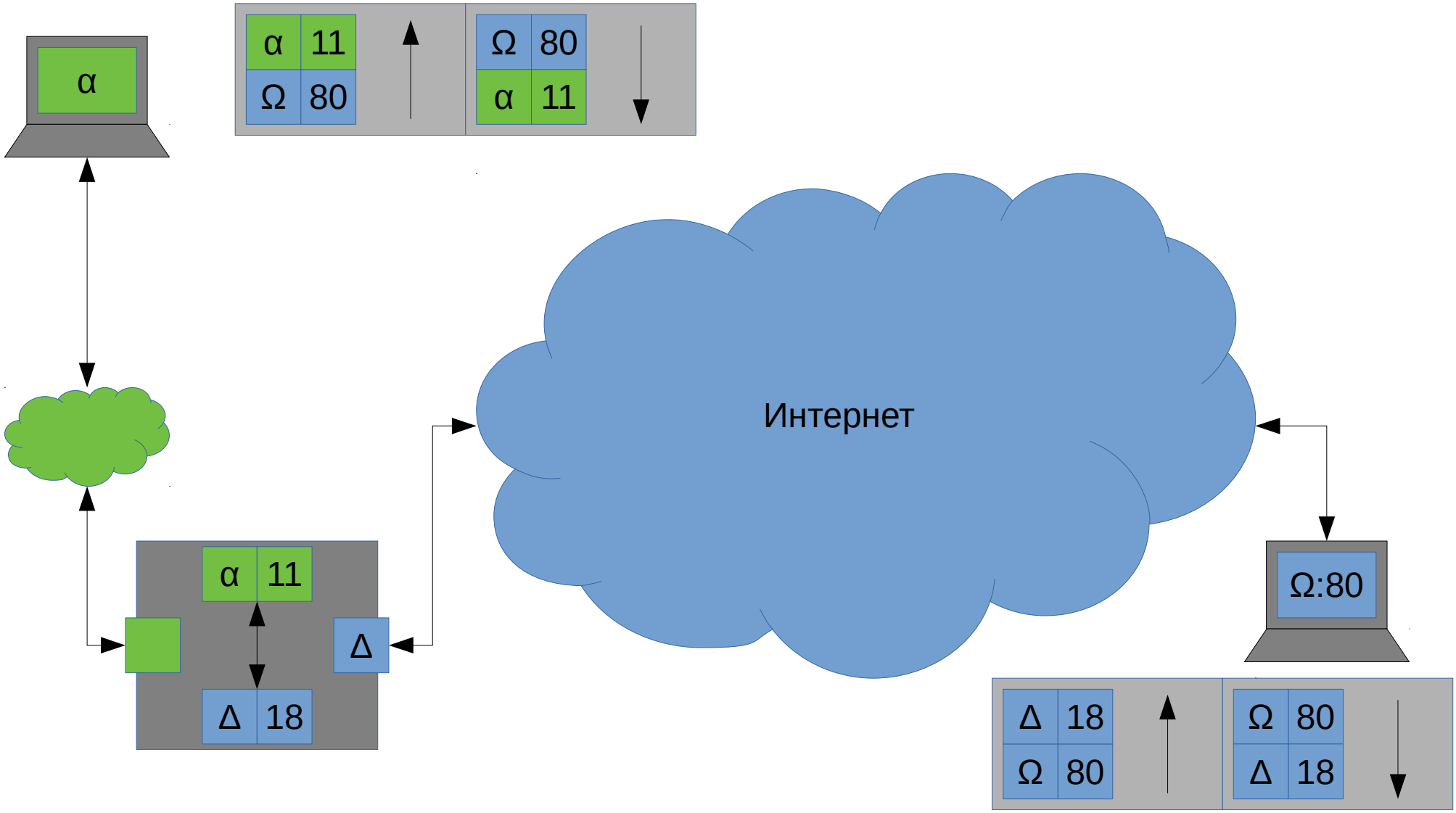
NAT



NAT



NAT



Протокол IPv6

Отличается от IPv4 использованием 128-битного адреса вместо 32-битного.

Адрес записывается как 8 групп по 4 шестнадцатеричные цифры.

```
2001:0db8:11a3:09d7:1f34:8a2e:07a0:765d
```

Допускается сокращение идущей подряд группы нулей.

```
2001:0000:0000:0000:0000:0000:0000:07a0
2001::07a0
```

Емкость адресов IPv6 составляет порядка 10^{38} адресов. Этого количества достаточно для многократного покрытия всех вычислительных потребностей.

В настоящее время IPv6 доступен на любом современном сетевом оборудовании и активно используется интернет-провайдерами.

Протокол IPv6

Специальные адреса в IPv6:

- :: — нулевой адрес, соответствует всем локальным устройствам.
- ::1 — адрес интерфейса обратной связи, аналогичен подсети 127.0.0.0/8 в IPv4.
- ::FFFF:xxxx:xxxx — адреса обратной совместимости с IPv4. Младшие байты соответствуют адресу IPv4.
- 2001:0db8:: — адрес для применения в программной документации.

Для обозначения маски подсети в IPv6 применяют только вариант с косой чертой:

- 2001:0db8::1/120

Система доменных имён

Очевидное решение для облегчения запоминания адресов серверов — присвоение серверу имени, легко запоминаемого человеком. Например, адресу 127.0.0.1 соответствует имя localhost.

Возможно использование локальной базы данных таких имён (файл hosts), либо сервера с такой базой (nameserver).

DNS (Domain Name System) — распределенная система, обеспечивающая присвоение имён серверам в сети интернет.

Домен — это имя в дереве имён вместе с подчиненными ему узлами.

Отдельные уровни доменных имен разделяются точкой.

Система доменных имён

Полное доменное имя, FQDN (Fully Qualified Domain Name) — это имя сервера вместе со всеми вышестоящими доменными именами.

Пример: www.example.com

- **www** — домен третьего уровня, обозначает конкретный сервер
- **example** — домен второго уровня. Сервер example.com является полноценным узлом сети, но при этом еще отвечает за все нижележащие доменные имена.
- **com** — домен первого уровня. Обслуживается национальными или региональными регистраторами.

Существует безымянный корневой домен нулевого уровня. Полный адрес: **www.example.com.**

Система доменных имён

Каждый узел DNS может производить преобразование между IP-адресом и доменным именем.

Ключевая особенность DNS, обеспечивающая децентрализацию — делегирование.

База данных DNS содержит следующие поля:

- Имя (NAME) — доменное имя, для которого предоставлена запись
- Тип записи (TYPE) — тип хранимых данных в базе.
- Сами данные (RDATA).

Система доменных имён

Некоторые из типов данных в DNS:

- A — хранит адрес в формате IPv4.
- AAAA — хранит адрес в формате IPv6.
- CNAME — каноническое имя, используется для перенаправления.
- MX — Mail Exchange, адрес почтового сервера для данного домена.
- NS — Name Server, адрес сервера, обслуживающего данный домен.
- PTR — запись для определения обратного соответствия. Для IP-адреса 192.0.34.164 запрос 164.34.0.192.in-addr.arpa вернет его каноничесоке имя referrals.icann.org.

Punycode

Ограничение системы доменных имен — это ограниченный набор символов, допустимых в поле NAME: строчные символы латинского алфавита, цифры и дефис.

Поддержка национальных доменных имен вида «пример.рф» осуществляется при помощи кодировки punycode. Она позволяет однозначно преобразовать символы Unicode в ограниченный набор символов доменных имен и обратно.

Адрес «пример.рф» в виде punycode выглядит как «xn--e1afmkfd.xn--p1ai».

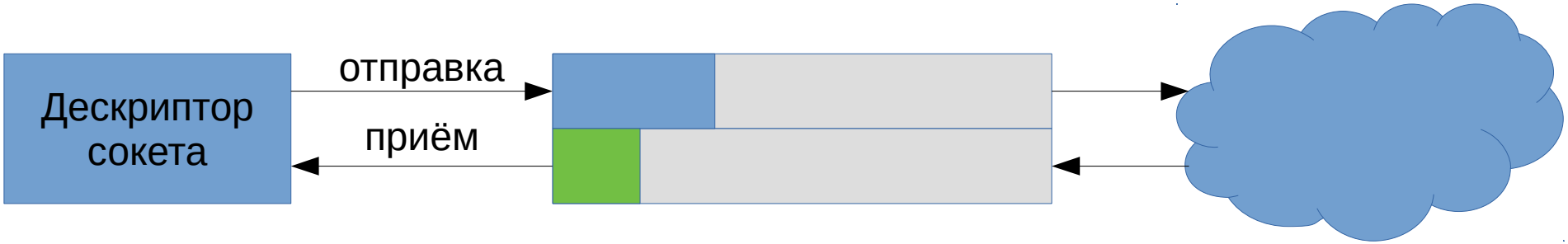
Сокет

Программный интерфейс для доступа к сетевым ресурсам носит название **сокет** (от англ socket — разъем).

Сокет является аналогом файлового дескриптора.

В установленном соединении сокету соответствует IP-адрес и порт локального и удаленного подключения.

Сокеты работают асинхронно: за каждым сокетом закрепляется пара буферов для хранения отправляемых и принятых данных. Операционная система автоматически обрабатывает эти буферы, прикладное ПО лишь может копировать данные в них и обратно.



Сокеты в C++

В среде Unix сокеты являются частью ядра системы и доступны путем подключения заголовочного файла sys/socket.h

```
#include <sys/socket.h>
```

В среде MS Windows сокеты реализованы в библиотеке winsock2, которую необходимо подключать и инициализировать.

```
LIBS += -lws2_32                                # В файле .pro
```

```
#include <winsock2.h>

int main(int argc, char* argv[])
{
  WSADATA wsadata;
  WSAStartup(MAKEWORD(2,2), &wsadata);
}
```

Сокеты в C++

Сеть является потенциально ненадежной, поэтому каждый вызов функции может закончиться ошибкой. При штатной работе код возврата функций сокета неотрицателен. При ошибке возвращается отрицательное значение. В Unix следует запросить значение переменной **errno** для уточнения ошибки. В MS Windows — вызвать функцию **WSAGetLastError()**.

```
int err = someSocketFunc(...);
if(err < 0)
{
    int code = WSAGetLastError();
    // Обработка ошибки
}
```

Справочное руководство по сетевым функциям содержит информацию о порождаемых ошибках.

Сокеты в C++

Для создания сокета необходимо вызвать функцию socket, передав ей в качестве аргумента его адресное семейство (протокол сетевого уровня), тип сокета (протокол транспортного уровня) и конкретное название протокола (0 для выбора протокола по умолчанию). Следующий код создает сокет TCP/IP:

```
SOCKET s;
s = socket(AF_INET, SOCK_STREAM, 0);
if (s == INVALID_SOCKET)
{
    cout << "Невозможно создать сокет" << endl;
}
```

Сокеты в C++

После создания необходимо подключить сокет к удаленному узлу. Для хранения адреса в памяти используется структура sockaddr.

Совместимая с ней структура sockaddr_in хранит IP-адрес и порт соединения TCP/IP. Можно заполнить ее следующим образом:

```
#include <ws2tcpip.h>
char* hostname = "localhost";
char* port = "5678";
int err;
sockaddr_in addr;
struct addrinfo *result = NULL;
err = getaddrinfo(hostname, port, NULL, &result);
if(err == 0)
    memcpy(&addr, result->ai_addr, sizeof(addr));
freeaddrinfo(result);
```

Сокеты в C++

При наличии IP-адреса и порта в структуре sockaddr_in можно вызвать функцию **connect** для подключения к удаленному узлу.

Сокету автоматически будет присвоен адрес и случайный свободный локальный порт на текущем компьютере.

В среде Qt следует использовать вызов connect из безымянного пространства имен, чтобы не создавать конфликт с методом connect из Qt.

```
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in addr;
// Заполнение addr
int err = ::connect(s, (sockaddr*)&addr, sizeof(addr));
```

Сокеты в C++

После успешного подключения становятся доступны функции **send** и **recv** для отправки и приема данных соответственно.

```
char buf[128];
int err = recv(s, buf, 128, 0);
if(err < 0)
{
    cout << "Unable to receive data" << endl;
    return;
}

err = send(s, buf, err, 0);
if(err < 0)
{
    cout << "Unable to send data" << endl;
    return;
}
```

Сокеты в C++

Аргументы send и recv:

- сокет
- начальный адрес массива данных
- количество байт (отправляемое или доступное для приема)
- флаги (всегда 0).

Возвращаемое значение **send** — это количество отправленных байт, либо число меньше 0 в случае ошибки.

Возвращаемое значение **recv** — это количество принятых байт в случае успешной операции. Если recv вернул 0, значит удаленный узел штатно завершил соединение. В случае ошибки recv вернет число меньше 0.

Сокеты в C++

При отсутствии данных в приемном буфере, recv заблокирует работу приложения до появления данных. При этом ОС приостановит процесс, и приложение не будет потреблять ресурсы до фактического прихода данных. Однако в графическом приложении это вызовет зависание интерфейса.

Возможный вариант решения — периодический опрос сокета на предмет наличия данных:

- **Неблокирующий режим.** При отсутствии данных в приемном буфере, функция recv вернется сразу, и будет установлен код ошибки WSAEWOULDBLOCK.
- **Таймаут.** При невозможности принять заданный объем данных в течение заданного времени, функция recv установит код ошибки WSAETIMEDOUT.

Сокеты в C++

Включение неблокирующего режима

```
u_long m=1;
ioctlsocket(s,FIONBIO,&m);
```

Отключение неблокирующего режима (по умолчанию)

```
u_long m=0;
ioctlsocket(s,FIONBIO,&m);
```

Задание таймаута для функции чтения

```
int timeout = 200; // таймаут в миллисекундах
setsockopt(s, SOL_SOCKET, SO_RCVTIMEO,
           (char*)&timeout, sizeof(timeout));
```

Сокеты в C++

Отключение сокета

```
SOCKET s = socket(AF_INET, SOCK_STREAM, 0);
// ...
shutdown(s, SD_BOTH);
closesocket(s);
s = 0;
```

После отключения можно использовать тот же дескриптор для создания нового сокета и нового соединения.

Сокеты в python

Сокеты python разработаны поверх сокетов C и предлагают во многом аналогичный интерфейс. Работа с сокетами обеспечивается модулем socket

```
import socket
```

Создание сокета происходит путем вызова функции **socket** из данного модуля.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Параметры по умолчанию создают сокет TCP/IP.

```
s = socket.socket()
```

Для закрытия сокета используется метод close.

```
s.close()
s = None
```

Сокеты в python

Обработка ошибок сокетов происходит при помощи исключений.

```
try:
    # socket functions
except socket.herror as he:
    # Ошибка определения адреса
    print(he)
except socket.gaierror as ge:
    # Ошибка определения адреса
    print(ge)
except socket.timeout as te:
    # Превышено время ожидания (таймаут)
    print(te)
except socket.error as e:
    # Все остальные ошибки
    print(e)
```

Сокеты в python

Для подключения сокета используется метод connect. Его аргумент — тьюпл, содержащий адрес и порт удаленного узла. Адрес может быть представлен в виде доменного имени.

```
s = socket()
s.connect(("example.com", 8080))
```

После подключения становятся доступны методы send и recv.

Аргумент send — байтовый массив (bytearray или bytes). Возвращаемое значение — фактическое количество отправленных байт.

Аргумент recv — количество байт, которое следует принять за одну операцию чтения. Возвращаемое значение — байтовый массив с данными. Возврат пустого массива означает штатное отключение удаленного узла.

Сокеты в python

```
s = socket.socket()
s.connect(("example.com", 8080))
try:
    request = b"<....>"
    s.send(request)
    while True:
        data = s.recv(256)
        print(data)
        if not data:
            break
except socket.error:
    s.close()
```

Сокеты в python

По умолчанию сокеты в python работают в блокирующем режиме с бесконечным таймаутом. Вызов метода recv при отсутствии данных в сокете заблокирует приложение до появления данных.

Метод **settimeout** позволяет задать таймаут операции в секундах. Если по окончании таймаута никакие данные не будут приняты, метод recv сгенерирует исключение.

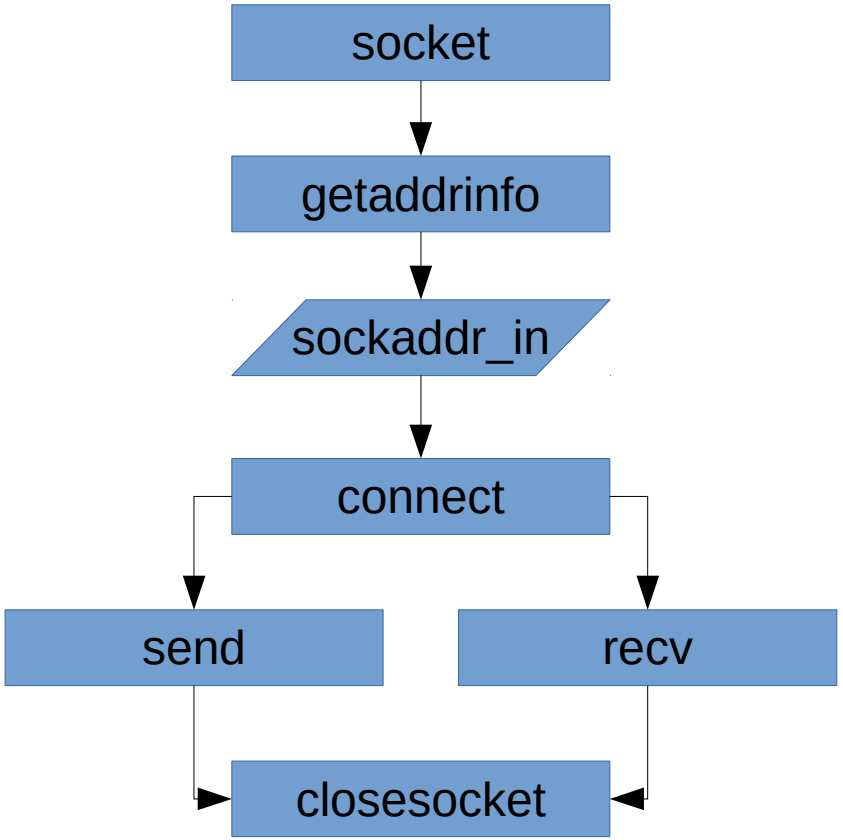
```
sock.settimeout(0.2) # Таймаут 0.2 секунды

try:
    s = sock.recv(2048)
except socket.timeout:
    # В сокете нет данных. Повторим чтение позже
    pass
```

Диаграмма работы TCP-клиента

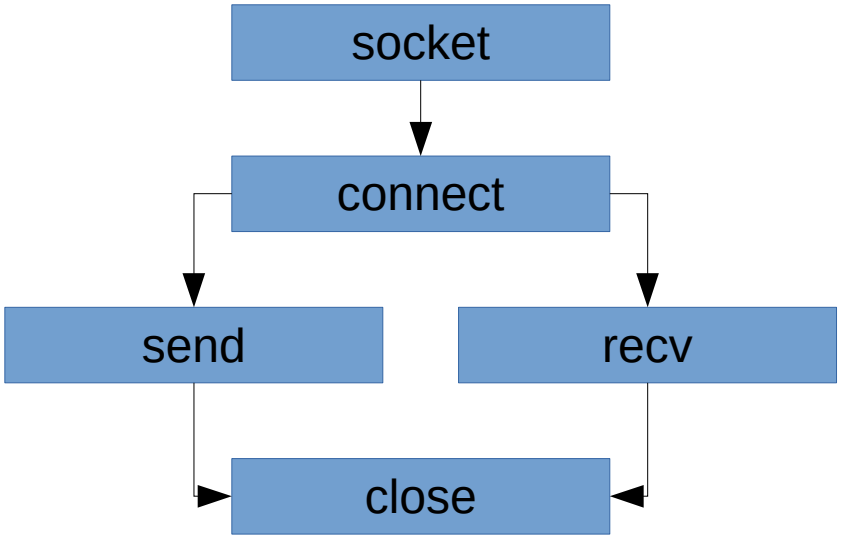
setsockopt
ioctlsocket
WSAGetLastError

C++



Python

try ... except
socket.error
settimeout



TCP-сервер

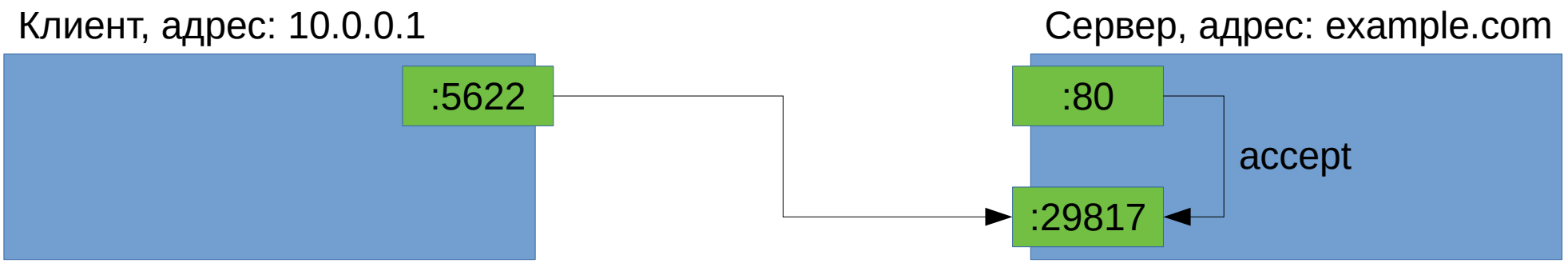
TCP-сервер использует специальный тип сокета — **слушающий сокет**. Такой сокет используется только для обмена служебными пакетами и не предназначен для передачи данных.

При появлении входящего подключения, сервер создает новый сокет со случайным свободным портом и переключает клиента на него, после чего обмен данными идет через новый сокет.

TCP-сервер

TCP-сервер использует специальный тип сокета — **слушающий сокет**. Такой сокет используется только для обмена служебными пакетами и не предназначен для передачи данных.

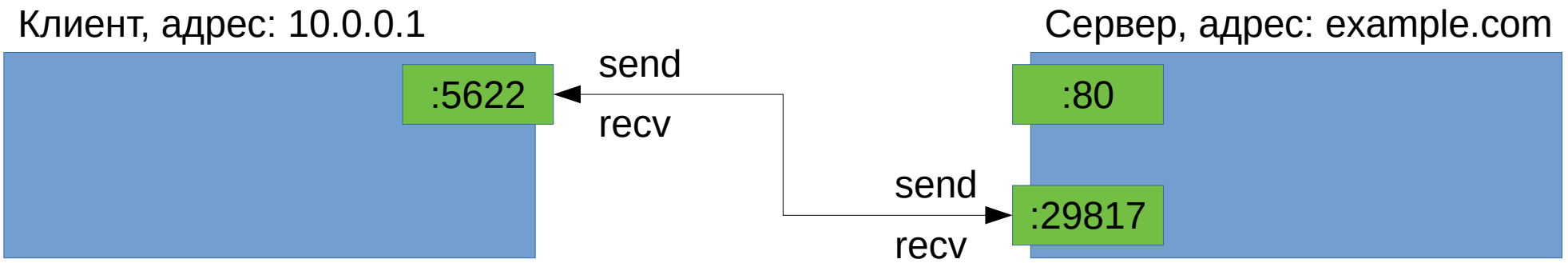
При появлении входящего подключения, сервер создает новый сокет со случайным свободным портом и переключает клиента на него, после чего обмен данными идет через новый сокет.



TCP-сервер

TCP-сервер использует специальный тип сокета — **слушающий сокет**. Такой сокет используется только для обмена служебными пакетами и не предназначен для передачи данных.

При появлении входящего подключения, сервер создает новый сокет со случайным свободным портом и переключает клиента на него, после чего обмен данными идет через новый сокет.



TCP-сервер

Для создания слушающего сокета необходимо сначала привязать его к определенному адресу, используя **bind**.

После занятия адреса, становится доступна функция **listen**, запускающая прослушивающий сокет. Аргумент функции — максимальная длина очереди.

```
// C++
sockaddr_in local;
local.sin_family = AF_INET;
local.sin_port = htons(port);
local.sin_addr.s_addr = INADDR_ANY;
bind(s, (sockaddr*)&local, sizeof(local));
listen(s, 10);

# python
s.bind(("", port))
s.listen(10)
```

TCP-сервер

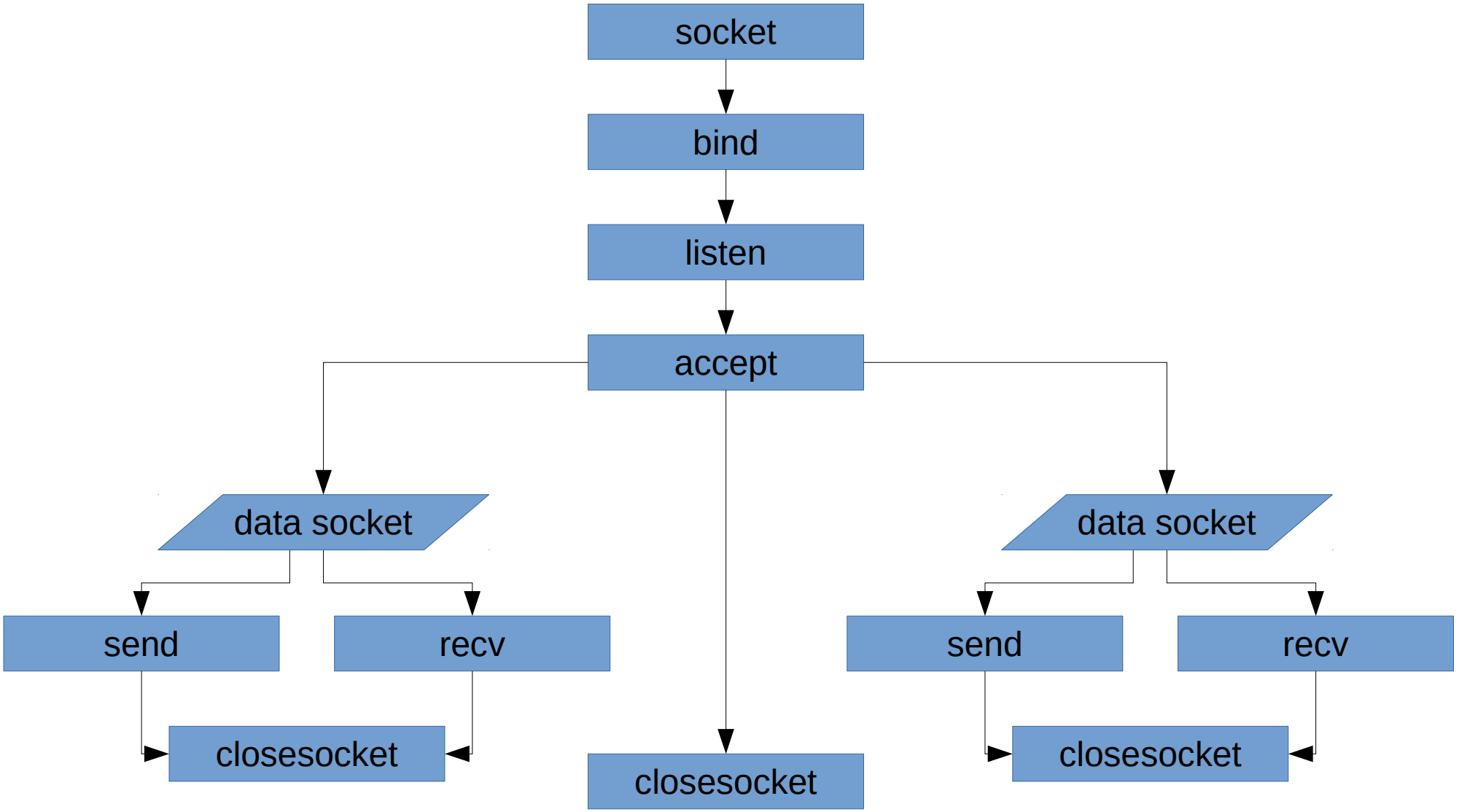
Прием входящего соединения и определение адреса подключившегося узла происходит при помощи функции **accept**.

```
// C++
sockaddr_in remote;
int r_len;
SOCKET ds = accept(s, (sockaddr*)&remote, &r_len);
cout << "Connection from " << inet_ntoa(remote.sin_addr)
     << ":" << ntohs(remote.sin_port);

# python
ds, addr = s.accept()
print("Connection from " + str(addr))
```

Полученный в результате сокет ds пригоден для обмена данными с клиентом.

Диаграмма работы TCP-сервера



Сокеты в Qt

Библиотека Qt предлагает свою реализацию сетевых интерфейсов — модуль network. Подключение модуля осуществляется в проектном файле.

```
QT += network
```

После этого становятся доступны классы **QTcpSocket**, **QUdpSocket** и **QTcpServer**.

В Python необходимо импортировать данные классы из соответствующего модуля.

```
from PyQt5.QtNetwork import QTcpSocket, QTcpServer
from PyQt5.QtNetwork import QUdpSocket
```

Сокеты в Qt

Сокеты Qt являются классами со своим набором методов.

Для подключения к удаленному узлу используется метод **connectToHost**. Аргументы — адрес и порт удаленного узла.

```
QTcpSocket s;       // C++
s = QTcpSocket()    # Python
s.connectToHost("example.com", 8080);
```

Для сокета всегда доступны методы **localAddress** и **localPort**, возвращающие локальные параметры сокета. Для подключенного сокета также доступны методы **peerAddress** и **peerPort**.

```
print(s.localAddress().toString(), s.localPort())
```

Для отключения сокета существует метод **disconnectFromHost**.

Для закрытия сокета следует использовать метод **close**.

Сокеты в Qt

Чтение и запись данных осуществляются методами **read** и **write**.

В C++ они могут принимать и возвращать QByteArray, либо работать с сырыми указателями. В Python используется bytearray.

```
QByteArray ba = s.read(256);
char* msg = "Hello world";
s.write(msg, strlen(msg));
```

```
ba = s.read(256);
msg = "Hello world";
s.write(msg.encode('utf-8'));
```

Сокеты в Qt

Сокеты Qt работают полностью асинхронно и порождают сигналы, обработка которых позволяет работать с сетью, не блокируя интерфейс приложения:

- **connected** — порождается после подключения сокета.
- **readyRead** — порождается, когда в сокете готова очередная порция данных для чтения. Можно забрать их вызвав метод **readAll**.
- **error** — порождается при возникновении ошибки в работе сокета. Метод сокета **errorString** возвращает строковое описание ошибки.
- **disconnected** — порождается при штатном отключении сокета.

Сокеты в Qt

```
class MainWindow: public QMainWindow
{
  QTcpSocket s;
public:
  MainWindow(QWidget* parent = nullptr)
  {
    connect(&s, SIGNAL(readyRead()), SLOT(ready()));
    s.connectToHost("example.com", 8080);
  }

private slots:
  void ready()
  {
    qDebug() << s.readAll();
  }
}
```

Сокеты в Qt

TCP-сервер в Qt реализован в виде класса **QTcpServer**. Работа с ним сводится к следующему:

- Создать объект QTcpServer.

```
QTcpServer srv;       // C++
srv = QTcpServer()    # Python
```

- Добавить обработчик сигнала **newConnection**.
- Запустить сервер методом **listen**.

```
srv.listen(QHostAddress::Any, port);
```

- В обработчике сигнала newConnection получить QTcpSocket, соединенный с клиентом, методом **nextPendingConnection**.

```
QTcpSocket* ds = srv.nextPendingConnection();
```

- По окончании работы с сервером следует закрыть его методом **close**. Это не прекратит работу уже подключенных через него сокетов. При необходимости их следует закрыть вручную.