**Лекция 2**

Итак, мы собираемся написать наше первое распределенное приложение. Способ, с помощью которого мы собираемся это сделать, - самый простой. Раз нам нужно передавать данные между компонентами нашего приложения по сети, следовательно, этому нужно научиться. Первый способ, который мы используем для решения задачи, будет основываться на входящем в состав пакета java.net API. Этот пакет предоставляет возможность для реализации сетевого взаимодействия с применением одного из двух транспортных протоколов: UDP и TCP.

Архитектура протоколов TCP/IP, известная как набор протоколов TCP/IP, возникла в результате исследований в области протоколов и разработок, выполнявшейся в экспериментальной сети с коммутацией пакетов под названием ARPANET, которая была основана Управлением перспективных исследовательских программ Министерства обороны США ( DARPA )[[3.26](https://www.intuit.ru/studies/courses/633/489/literature#literature.26)]. Этот набор протоколов состоит из большого собрания протоколов, изданных Координационным советом по сети Internet (IAB) в качестве стандартов для Internet.

В общем можно сказать, что в обмене информацией всегда принимают участие три агента: приложения, узлы (компьютеры) и сети. К приложениям относятся программы, предназначенные, например, для передачи файлов или электронной почты. Приложения, о которых идет речь, являются распределенными приложениями, включающими в себя обмен данными между двумя и более компьютерными системами. Компьютеры соединены между собой в сети, и предназначенные для обмена данные передаются по сети от одного компьютера к другому. Таким образом, передача данных от одного приложения другому (или от одного компонента другому в рамках одного распределенного приложения) включает в себя, во-первых, получение данных тем узлом, на котором находится приложение-адресат, и, во-вторых, получение данных именно тем выполняющимся на узле-адресате приложением, которому они предназначены (одновременно на узле могут выполняться несколько приложений).

Ввиду этого в задаче обмена информацией выделяют пять основных относительно независимых уровней:

* физический уровень;
* уровень доступа к сети (сетевой);
* межсетевой уровень;
* транспортный уровень;
* уровень приложений.

На физическом уровне находится физический интерфейс между устройством передачи данных и передающей средой. На этом уровне задаются характеристики передающей среды, природа сигналов, скорость передачи данных и т.д.

Уровень доступа к сети связан с обменом данными между конечной системой (рабочей станцией, сервером и т.д.) и сетью, к которой подсоединена эта система. Узел-отправитель должен передать в сеть адрес узла-адресата, чтобы сеть могла направить данные по месту требования.

Уровень доступа к сети рассматривается в связи с доступом к сети и маршрутизацией данных между двумя подключенными к сети оконечными системами. В тех случаях, когда устройства подключены к разным сетям, нужны дополнительные процедуры, позволяющие данным переходить из одной сети в другую. Такие функции относятся к межсетевому уровню. На этом уровне функции межсетевой маршрутизации предоставляются с помощью Internet -протокола ( IP ). Internet -протокол может быть реализован не только в конечных системах, но и в маршрутизаторах.

Независимо от природы приложений обмен данными должен быть надежным, т.е. хотелось бы иметь уверенность в том, что все данные попали к приложению-адресату и что эти данные получены в том порядке, в котором они были отправлены. Механизмы обеспечения надежности, по сути, независимы от природы приложений, таким образом, имеет смысл выделить такие механизмы в один общий уровень, совместно используемый всеми приложениями. Этот уровень называется транспортным. Чаще всего для него применяется протокол управления передачей ( Transmission Control Protocol - TCP ).

Наконец, в уровень приложений заложена логика, необходимая для обеспечения работы различных пользовательских приложений. Приложению каждого вида нужен отдельный модуль, специально предназначенный для этого приложения. Именно разработкой уровня приложения мы в дальнейшем и займемся.

Итак, чтобы обмен информацией был возможен, каждый элемент системы должен иметь уникальный адрес. Фактически, нужно указать два уровня адресации. Каждый узел сети должен обладать своим уникальным глобальным сетевым адресом, это позволит доставить данные соответствующему узлу. Каждый процесс (компонент) на узле должен иметь адрес, который был бы уникальным в пределах этого узла, что даст возможность транспортному протоколу ( TCP ) доставить данные нужному процессу. Этот адрес - адрес процесса - в терминологии протоколов семейства TCP/IP называют портом.

Для большинства приложений, выполняющихся в рамках архитектуры протокола TCP/IP, протоколом транспортного уровня выступает TCP. Этот протокол обеспечивает надежное соединение для передачи данных от одного приложения к другому. Кроме него существует еще один широко используемый протокол транспортного уровня, входящий в набор протоколов TCP/IP: пользовательский протокол датаграмм ( User Datagram Protocol - UDP ). Протокол UDP предоставляет сервис без установления соединения, предназначенный для процедур на уровне приложения; он не гарантирует доставку, сохранение последовательности при приеме переданных пакетов или защиту от их дублирования. Он позволяет процессу отправлять сообщения другим процессам, с помощью минимального протокольного механизма. Протокол UDP выполняет крайне ограниченный набор функций, так как работает без установления соединения. По сути, он добавляет к протоколу IP некоторые возможности адресации портов.

**UDP**

Начнем мы с API, использующего протокол UDP. UDP является протоколом, применяющим для передачи датаграммы. Как уже говорилось, этот протокол не является надежным, поскольку сообщения, передаваемые с его помощью, могут теряться или приходить в другой последовательности, отличной от последовательности их отправки. Таким образом, для обеспечения надежной передачи необходимо организовывать надстройку над этим протоколом, обеспечивающую, например, нумерацию пакетов, повторную передачу пакетов при истечении времени ожидания и т.д. Длина одного сообщения (одной датаграммы) при использовании этого протокола ограничена 65536 байтами (причем многие реализации вообще ограничивают размер датаграммы 8К), в случае необходимости пересылки порции данных большего размера они должны быть разбиты на куски отправителем и снова собраны получателем. Передача сообщения - не блокирующая, прием - блокирующий с возможностью прерывания по истечении времени ожидания.

Для работы с UDP в пакете java.net определены следующие классы.

* **DatagramPacket** (датаграмма). Конструктор этого класса принимает массив байт и адрес процесса-получателя ( IP -адрес узла и порт). Класс предназначен для представления единичной дата-граммы (сообщения). Этот класс используется как для создания сообщения с целью последующей его отправки, так и при приеме сообщения (функция приема возвращает экземпляр этого класса).
* **DatagramSocket**.Предназначен для посылки/приема UDP -дата-грамм. Один из конструкторов принимает в качестве аргумента порт, с которым связывается сокет, другой конструктор, без аргументов, задействует в качестве порта первый попавшийся свободный порт. Класс имеет методы send и receive, для, соответственно, передачи и приема датаграмм. Метод setSoTimeout устанавливает тайм-аут для операций сокета.

Ниже приведены две простые программы, использующие рассмотренные механизмы для организации взаимодействия.

**UDPClient**

Первая программа создает сокет, соединяется с сервером (порт 6789), пересылает ему сообщение и ждет ответа.

1 import java.net.\*;

2 import java.io.\*;

3 public class UDPClient{

4 public static void main(String args[]){

5 // аргументы - сообщение и адрес сервера

6 try {

7 DatagramSocket aSocket = new DatagramSocket(); // create socket

8 byte [] message = args[0].getBytes();

9 InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup

10 int serverPort = 6789;

11 DatagramPacket request =

12 new DatagramPacket(message, args[0].length(), aHost, serverPort);

13 aSocket.send(request);

14 //send message

15 byte[] buffer = new byte[1000];

16 DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

17 aSocket.receive(reply);

18 //wait for reply

19 System.out.println("Reply: " + new String(reply.getData()));

20 aSocket.close();

21 } catch (SocketException e){ System.out.println("Socket: " + e.getMessage());

22 // socket creation failed

23 } catch (IOException e){ System.out.println("IO: " + e.getMessage());

24 // ошибка при приеме

25 }

26 }

27 }