

## Тема: Basics of Java Network Programming (Part 3)

### План занятия:

1. Передача информации группе хостов
  - Широковещательная передача (broadcasting)
  - Многоадресное взаимодействие (multicasting)
  - Групповые адреса (Multicast Addresses) и группы
  - Клиенты и серверы
  - Маршрутизаторы и маршрутизация
2. Параметры сокета DatagramSocket
  - Параметр SO\_REUSEADDR
  - Параметр SO\_BROADCAST
3. Работа с многоадресными сокетами
  - Конструкторы
  - Взаимодействие с многоадресной группой
  - Присоединение к группам многоадресной рассылки
  - Выход из группы и закрытие соединения
  - Отправка многоадресных сообщений
  - Режим обратной петли (loopback)
  - Сетевые интерфейсы
4. Примеры
  - Простой полный пример
  - Второй пример
  - Третий пример
  - Еще два примера

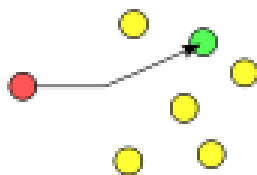
### Литература

1. Harold E. R. *Java Network Programming*: - O'Reilly, 2005 (2013) – 735 p.
2. Trail: Custom Networking:  
<https://docs.oracle.com/javase/tutorial/networking/>
3. Роберт Орфали, Дан Харки. *Java и CORBA в приложениях клиент-сервер*. Издательство: Лори, 2000 - 734 с.
4. Патрик Нимейер, Дэниэл Леук. *Программирование на Java* (Исчерпывающее руководство для профессионалов). М.: Эксмо, 2014

### Передача информации группе хостов

Те сокеты, которые мы рассмотрели на предыдущих лекциях, были *одноадресными* (*unicast*): они обеспечивают связь только между двумя четко определенными конечными точками - хостами; был один отправитель

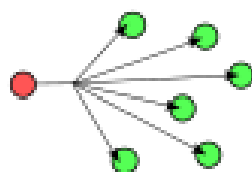
сообщения и один получатель сообщения. Хотя хосты могут меняться ролями, в любой момент времени легко определить, кто есть кто.



Для решения многих задач такой модели «*point-to-point*» вполне достаточно, но существуют задачи, для решения которых требуется другая модель. Иногда требуется отправить одно и то же сообщение на все устройства, подключенные к сети и имеющие *IP*-адрес в определенном диапазоне, или на группу хостов, которые расположены в разных сегментах сети. Для решения такого рода задач есть стандартные подходы, называемые широковещание (англ. *broadcasting*) и групповая рассылка (*multicasting*).

### Широковещательная передача (*broadcasting*)

Широковещательная (*broadcast*) связь - это когда один отправитель передает данные на все устройства, подключенные к сети, имеющие *IP*-адрес в определенном диапазоне. Эта передача может быть нацелена на все локальные устройства подсети, все узлы в локальной сети и т.д.



В *TCP/IP* широковещание (*broadcast*) возможно только в пределах одного сегмента сети (*L2* или *L3*). Однако пакеты данных могут быть посланы из-за пределов сегмента, в который будет осуществлено.

Широковещательный адрес — условный (не присвоенный никакому устройству в сети) адрес, который используется для передачи широковещательных пакетов в компьютерных сетях. Впервые технология использования широковещательных адресов в *IP* сетях была предложена в 1982 году Робертом Гурвицем (*Robert Gurwitz*) и Робертом Хинденом (*Robert Hinden*).

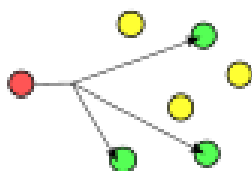
В зависимости от уровня модели *OSI* различают несколько видов широковещательных адресов. В *IP*-сетях широковещательные адреса формируются следующим образом: к адресу подсети прибавляется побитовая инверсия маски подсети (то есть все биты адреса подсети, соответствующие

нулям в маске, устанавливаются в «1»). Например, если адрес сети равен 192.168.0.0, маска подсети 255.255.255.0, то широковещательный адрес будет 192.168.0.255. По сути, широковещательным IP адресом является последний адрес в подсети. Если сеть состоит из одного адреса /32, то она не имеет широковещательного адреса. Адрес 255.255.255.255 является ограниченным широковещательным адресом. На пакет с таким адресом назначения должны ответить все хосты из любых подсетей в пределах L2 домена.

К передаче пакетов на широковещательные адреса следует относиться с предельной осторожностью. Некорректное использование может привести к нарушению работоспособности как отдельного сегмента, так и сети в целом (см. широковещательный шторм). Исходя из соображений безопасности и обеспечения максимальной пропускной способности сети, на шлюзах может быть установлен запрет транзита пакетов на широковещательные адреса.

### Многоадресное взаимодействие (*multicasting*)

Многоадресная рассылка шире, чем одноадресная (*unicast*), связь точка-точка (*point-to-point*), но более узкая и более целенаправленная, чем широковещательная (*broadcast*) связь. Групповая рассылка отправляет данные с одного хоста на множество разных хостов, но не всем; данные отправляются только тем клиентам, которые проявили интерес, присоединившись к определенной группе многоадресной рассылки.



В любой момент времени хост может покинуть такую группу и перестанет получать многоадресные сообщения. Для реализации такой модели существует *многоадресный сокет*, который отправляет копию данных в местоположение (или группу местоположений), расположенное рядом со сторонами, которые объявили интерес к данным. В лучшем случае данные дублируются только тогда, когда они достигают локальной сети, обслуживающей заинтересованных клиентов: данные пересекают *Интернет* только один раз. Это идеальная ситуация, более реально, что несколько идентичных копий данных пересекают *Интернет*; но благодаря тщательному выбору точек, в которых дублируются потоки, нагрузка на сеть минимизируется. Главное, что и программисты, и сетевые администраторы не несут ответственности за выбор точек, в которых данные дублируются, интернет-роутеры справляются со всем этим.

Когда речь идет о групповом вещании, первое, что вспоминается, - это аудио и видео трансляции. Действительно, *BBC* уже достаточно давно проводит многоадресное вещание, охватывающее как телевидение, так и радио. Другие возможности включают многопользовательские игры, распределенные файловые системы, массовые параллельные вычисления, многопользовательские конференции, репликацию базы данных, сети доставки контента и многое другое. Многоадресную рассылку можно использовать для реализации служб имен и служб каталогов, которые не требуют, чтобы клиент заранее знал адрес сервера; чтобы найти имя, хост мог бы направить свой запрос по известному адресу и подождать, пока ответ не будет получен от ближайшего сервера. *Apple Bonjour* (см. статью <https://ru.wikipedia.org/wiki/Bonjour>) и *Apache's River* (см. <https://river.apache.org/>) используют *IP*-адресацию для динамического обнаружения служб в локальной сети.

Многоадресная рассылка была разработана так, чтобы как можно более легко вписываться в *Интернет*. Большая часть работы выполняется маршрутизаторами и должна быть прозрачной для разработчиков приложений. Приложение просто отправляет пакеты датаграмм на многоадресный адрес, который принципиально не отличается от любого другого *IP*-адреса. Маршрутизаторы удостоверяются, что пакет доставлен всем хостам в группе многоадресной рассылки. Самая большая проблема заключается в том, что многоадресные маршрутизаторы еще не повсеместны; поэтому вам нужно специально узнавать, поддерживается ли в вашей сети многоадресная рассылка. На практике многоадресная рассылка гораздо чаще используется за брандмауэром в рамках одной организации, чем через глобальный *Интернет*.

Что касается самого приложения, то необходимо обратить внимание на дополнительное поле заголовка в датаграммах, называемое значением времени жизни (*TTL*, см. статью [https://ru.wikipedia.org/wiki/Time\\_to\\_live](https://ru.wikipedia.org/wiki/Time_to_live)). *TTL* - это максимальное количество маршрутизаторов, которое датаграмме разрешено пересекать. Как только пакет пересекает указанное количество маршрутизаторов, он отбрасывается. Многоадресная рассылка использует *TTL* в качестве специального способа ограничения расстояния, которое может пройти пакет.

## **Групповые адреса (*Multicast Addresses*) и группы**

Групповой адрес (адрес многоадресной рассылки, *multicast address*) – это общий адрес целой группы хостов, которая называется группой многоадресной рассылки (*multicast group*). Обсудим сначала понятие группового адреса (см. статью [https://en.wikipedia.org/wiki/Multicast\\_address](https://en.wikipedia.org/wiki/Multicast_address)). Групповые адреса *IPv4*

- это IP-адреса в группе *CIDR* (англ. *Classless Inter-Domain Routing*, см. статью [https://en.wikipedia.org/wiki/Classless\\_Inter-Domain\\_Routing](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing)) 224.0.0.0/4 (т.е. они находятся в диапазоне от 224.0.0.0 до 239.255.255.255). Все адреса в этом диапазоне имеют двоичные цифры 1110 в своих первых четырех битах. Многоадресные адреса *IPv6* находятся в группе *CIDR* ff00::/8 (т.е. все они начинаются с байта 0xFF или 11111111 в двоичном формате).

Подобно любому IP-адресу, групповой адрес может иметь имя (*hostname*). Например, групповому адресу 224.0.1.1 (адрес *Network Time Protocol distributed service – протокол сетевого времени*) присваивено имя ntp.mcast.net.

Многоадресная группа (*multicast group*, см. статью [https://en.wikipedia.org/wiki/IP\\_multicast](https://en.wikipedia.org/wiki/IP_multicast)) - это набор интернет-хостов, которые имеют один многоадресный адрес. Любые данные, отправленные на адрес многоадресной группы, передаются всем членам группы. Членство в многоадресной группе открыто: хосты могут свободно входить или выходить из группы в любое время. Группы могут быть постоянными или временными. Постоянным группам назначены адреса, которые остаются постоянными, независимо от того, есть ли в группе какие-либо члены. Однако большинство многоадресных групп являются временными и существуют только до тех пор, пока у них есть участники. Все, что вам нужно сделать, чтобы создать новую группу многоадресной рассылки, это выбрать некоторый адрес из диапазоне от 225.0.0.0 до 238.255.255.255, создать объект *InetAddress*, соответствующий этому адресу и начать отправлять ему данные.

*IANA* (от англ. *Internet Assigned Numbers Authority* — «Администрация адресного пространства Интернет», <https://www.iana.org/>, см. статью <https://ru.wikipedia.org/wiki/IANA>) отвечает за выдачу постоянных многоадресных адресов. Адреса назначаются по мере необходимости, и пока что специально назначены несколько сотен таких адресов. Локальные групповые адреса (*link-local multicast addresses*) начинаются с 224.0.0 (то есть эти адреса принадлежат диапазону от 224.0.0.0 до 224.0.0.255) и зарезервированы для протоколов маршрутизации и других низкоуровневых действий, таких как обнаружение шлюза и отчеты о членстве в группах. Например, адрес all-systems.mcast.net, 224.0.0.1, соответствует многоадресной группе, которая включает все системы в локальной подсети. Многоадресные маршрутизаторы никогда не пересылают дальше датаграммы с адресатами в этом диапазоне. В таблице 13-1 перечислены некоторые из этих назначенных адресов.

Таблица: Link-local multicast addresses

Domain name	IP address	Purpose
<i>BASE-ADDRESS.MCAST.NET</i>	<i>224.0.0.0</i>	The reserved base address. This is never assigned to any multicast group.
<i>ALL-SYSTEMS.MCAST.NET</i>	<i>224.0.0.1</i>	All systems on the local subnet.
<i>ALL-ROUTERS.MCAST.NET</i>	<i>224.0.0.2</i>	All routers on the local subnet.
<i>DVMRP.MCAST.NET</i>	<i>224.0.0.4</i>	All Distance Vector Multicast Routing Protocol (DVMRP) routers on this subnet.
<i>MOBILE-AGENTS.MCAST.NET</i>	<i>224.0.0.11</i>	Mobile agents on the local subnet.
<i>DHCP-AGENTS.MCAST.NET</i>	<i>224.0.0.12</i>	This multicast group allows a client to locate a Dynamic Host Configuration Protocol (DHCP) server or relay agent on the local subnet.
<i>RSVP-ENCAPSULATION.MCAST.NET</i>	<i>224.0.0.14</i>	RSVP encapsulation on this subnet. RSVP stands for Resource reSerVation setup Protocol, an effort to allow people to reserve a guaranteed amount of Internet bandwidth in advance for an event.
<i>VRRP.MCAST.NET</i>	<i>224.0.0.18</i>	Virtual Router Redundancy Protocol (VRRP) Routers
	<i>224.0.0.35</i>	<b>DXCluster</b> is used to announce foreign amateur (DX) stations.
	<i>224.0.0.36</i>	Digital Transmission Content Protection (DTCP), a digital restrictions management (DRM) technology that encrypts interconnections between DVD players, televisions, and similar devices.
	<i>224.0.0.37-224.0.0.68</i>	zeroconf addressing
	<i>224.0.0.106</i>	<b>Multicast Router Discovery</b>

Domain name	IP address	Purpose
	<i>224.0.0.112</i>	Multipath Management Agent Device Discovery
	<i>224.0.0.113</i>	Qualcomm's AllJoyn
	<i>224.0.0.114</i>	Inter RFID Reader Protocol
	<i>224.0.0.251</i>	<b>Multicast DNS</b> self assigns and resolves host names for multicast addresses.
	<i>224.0.0.252</i>	<b>Link-local Multicast Name Resolution</b> , a precursor of mDNS, allows nodes ot self-assign domain names strictly for the local network, and to resolve such domain names on the local network.
	<i>224.0.0.253</i>	<b>Teredo</b> is used to tunnel IPv6 over IPv4. Other Teredo clients on the same IPv4 subnet respond to this multicast address.
	<i>224.0.0.254</i>	Reserved for experimentation.



Таблица: Общие постоянные адреса многоадресной рассылки

Domain name	IP address	Purpose
<i>NTP.MCAST.NET</i>	224.0.1.1	The Network Time Protocol.
<i>NSS.MCAST.NET</i>	224.0.1.6	The Name Service Server.
<i>AUDIONEWS.MCAST.NET</i>	224.0.1.7	Audio news multicast.
<i>MTP.MCAST.NET</i>	224.0.1.9	The Multicast Transport Protocol.
<i>IETF-1-LOW-AUDIO.MCAST.NET</i>	224.0.1.10	Channel 1 of low-quality audio from IETF meetings.
<i>IETF-1-AUDIO.MCAST.NET</i>	224.0.1.11	Channel 1 of high-quality audio from IETF meetings.
<i>IETF-1-VIDEO.MCAST.NET</i>	224.0.1.12	Channel 1 of video from IETF meetings.
<i>IETF-2-LOW-AUDIO.MCAST.NET</i>	224.0.1.13	Channel 2 of low-quality audio from IETF meetings.
<i>IETF-2-AUDIO.MCAST.NET</i>	224.0.1.14	Channel 2 of high-quality audio from IETF meetings.
<i>IETF-2-VIDEO.MCAST.NET</i>	224.0.1.15	Channel 2 of video from IETF meetings.
<i>MLOADD.MCAST.NET</i>	224.0.1.19	MLOADD measures the traffic load through one or more network interfaces over a number of seconds. Multicasting is used to communicate between the different interfaces being measured.
<i>EXPERIMENT.MCAST.NET</i>	224.0.1.20	Experiments.
	224.0.23.178	JDP Java Discovery Protocol, used to find manageable JVMs on the network.
<i>MICROSOFT.MCAST.NET</i>	224.0.1.24	Used by Windows Internet Name Service (WINS) servers to locate one another.
<i>MTRACE.MCAST.NET</i>	224.0.1.32	A multicast version of traceroute.
<i>JINI-ANNOUNCEMENT.MCAST.NET</i>	224.0.1.84	JINI announcements.
<i>JINI-REQUEST.MCAST.NET</i>	224.0.1.85	JINI requests.
	224.0.1.143	Emergency Managers Weather Information Network.
	224.2.0.0-224.2.255.255	The Multicast Backbone on the Internet (MBONE) addresses are reserved for multimedia conference calls (i.e., audio, video, whiteboard, and shared web browsing between many people).
	224.2.2.2	Port 9875 on this address is used to broadcast the currently available MBONE programming. You can look at this with the X Window utility sdr or the Windows/Unix multikit program.
	239.0.0.0-239.255.255.255	Organization local scope, in contrast to TTL scope, uses different ranges of multicast addresses to constrain multicast traffic to a particular region or group of routers. For example, when a Universal Plug and Play (UPnP) device joins a network, it sends an HTTPU (HTTP over UDP) message to the multicast address 239.255.255.250 on port 1900. The idea is to allow the possible group membership to be established in advance without relying on less-than-reliable TTL values.

Постоянно назначенные групповые адреса, которые выходят за пределы локальной подсети, начинаются с 224.1. или 224.2. В таблице, расположенной ниже, перечислены некоторые из этих постоянных адресов. Несколько блоков адресов размером от нескольких десятков до нескольких

тысяч адресов также были зарезервированы для некоторых конкретных целей. Полный список адресов доступен на сайте [iana.org](http://iana.org), хотя там есть много уже не функционирующих сервисов, протоколов и компаний. Оставшиеся 248 миллионов адресов многоадресной рассылки могут временно использоваться всеми, кто в них нуждается. Многоадресные маршрутизаторы отвечают за то, чтобы две разные системы не пытались использовать один и тот же адрес одновременно.

## Клиенты и серверы

Когда хост хочет отправить данные в группу многоадресной рассылки, он должен поместить эти данные в датаграммы, предназначенные для многоадресной рассылки, которые, технически, являются *UDP* датаграммами, которые адресованы группе многоадресной рассылки. Данные, предназначенные для многоадресной рассылки отправляются по *UDP* протоколу. Это, вообще-то ненадежный протокол, но зато доставка данных может быть выполнена раза в три быстрее, по *TCP* протоколу с установлением соединения. Если разрабатывается многоадресное приложение, нужно решить, что следует делать при потере данных.

С точки зрения разработчика *Java* приложений основное отличие между многоадресной передачей и передачей с использованием обычных *UDP* сокетов заключается в том, что нужно выбрать значение *TTL* (*time-to-live*). Физически это параметр определяется одним байтом в *IP* заголовке, который может принимать значения в диапазоне от 1 до 255; его можно приблизительно интерпретировать, как количество маршрутизаторов, через которые может пройти пакет, прежде чем он будет отброшен. Всякий раз, когда пакет проходит через маршрутизатор, значение, хранящееся в его *TTL* поле, уменьшается как минимум на единицу; некоторые маршрутизаторы могут уменьшать *TTL* на два или более. Когда значение параметра *TTL* достигает нуля, пакет отбрасывается. По замыслу, *TTL* поле предназначалось для предотвращения петель по пути пакте, гарантируя, что все пакеты будут в конечном итоге отброшены; это препятствует неправильно настроенным маршрутизаторам бесконечно отправлять пакеты друг другу.

При организации многоадресной *IP*-рассылки значение поля *TTL* географически ограничивает многоадресную рассылку. Например, значение *TTL*, равное 16, ограничивает распространение пакета локальной областью, обычно одной организацией или, возможно, организацией и ее ближайшими соседями. Поле *TTL*, равное 127, разрешает отправлять пакет адресатам по всему миру. Допускаются промежуточные значения. Но, все-таки, нет точного способа сопоставить параметр *TTL* с географическим расстоянием. Как правило, чем дальше находится сайт, тем больше маршрутизаторов должен



пройти пакет, прежде чем достигнуть его. Пакеты с малыми значениями *TTL* не пройдут дальше, чем пакеты с большими значениями *TTL*. В таблице, приведенной чуть ниже, указаны некоторые приблизительные оценки, связывающие значения *TTL* и географическое расстояние. Но, следует помнить, что пакеты, адресованные многоадресной группе с адресами в диапазоне от 224.0.0.0 по 224.0.0.255, никогда не пересылаются за пределы локальной подсети, независимо от используемых значений *TTL*.

Таблица: Расчетные значения *TTL* для датаграмм, распространяющихся в континентальной части США.

Destinations	TTL value
The local host	0
The local subnet	1
The local campus—that is, the same side of the nearest Internet router—but on possibly different LANs	16
High-bandwidth sites in the same country, generally those fairly close to the backbone	32
All sites in the same country	48
All sites on the same continent	64
High-bandwidth sites worldwide	128
All sites worldwide	255

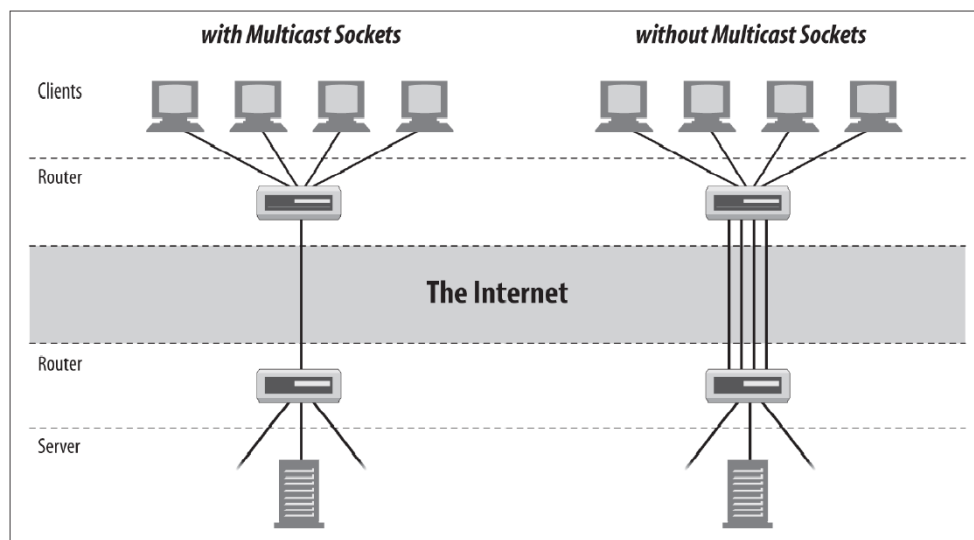
Как только данные упакованы в одну или несколько датаграмм, хост-отправитель запускает датаграммы в *Интернет*. Процесс очень похож на отправку обычных *одноадресных UDP*-датаграмм. Хост-отправитель начинает передачу многоадресной датаграммы в локальную сеть. Этот пакет сразу достигает всех членов многоадресной группы в одной подсети. Если поле *Time-To-Live* пакета больше 1, многоадресные маршрутизаторы локальной сети пересылают пакет в другие сети, в которых есть члены группы назначения. Когда пакет прибывает в один из конечных пунктов назначения, многоадресный маршрутизатор во внешней сети передает пакет на каждый обслуживаемый им хост, который является членом группы многоадресной рассылки. При необходимости многоадресный маршрутизатор передает пакет следующим маршрутизаторам; тем, которые расположены по пути между текущим маршрутизатором и всеми возможными назначениями.

Когда данные прибывает на хост, принадлежащий группе многоадресной рассылки, хост получает их так же, как и любые другие *UDP* датаграммы, с учетом того, что адрес назначения пакета может не совпадать с адресом принимающего хоста. Хост распознает, что датаграмма предназначена для него, потому что он принадлежит к группе многоадресной рассылки, которой адресована дейтаграмма. Принимающий хост просто должен прослушивать

соответствующий порт и быть готовым обработать датаграмму, как только она придет.

## Маршрутизаторы и маршрутизация

Рассмотрим одну из самых простых конфигураций многоадресной рассылки: один сервер отправляет одни и те же данные четырем клиентам, которые обслуживаются одним и тем же маршрутизатором.



Многоадресный сокет отправляет один поток данных через *Интернет* на маршрутизатор клиента; маршрутизатор дублирует поток и отправляет его каждому клиенту. Без многоадресных сокетов серверу пришлось бы отправлять четыре отдельных, но идентичных потока данных на маршрутизатор, который направляет каждый поток клиенту. Использование нескольких копий одного и того же потока для отправки одних и тех же данных нескольким клиентам значительно снижает пропускную способность.

Понятно, что реальные маршруты пакетов могут быть намного более сложными, включая большое количество иерархий маршрутизаторов. Но, назначение многоадресных сокетов остается тем же: независимо от того, насколько сложна сеть, одни и те же данные никогда не следует отправлять более одного раза по любому сегменту сети. И главное, программисту не нужно беспокоиться о проблемах маршрутизации. Нужно просто создать объект `MulticastSocket`, подключить сокет к группе многоадресной рассылки и вставить адрес группы многоадресной рассылки в `DatagramPacket`, который необходимо отправить. Маршрутизаторы и класс `MulticastSocket` позаботятся обо всем остальном.

Самое большое ограничение, накладываемое на многоадресную передачу - это наличие специальных многоадресных маршрутизаторов,

поддерживающих *IP*-адреса многоадресной рассылки. Многие ориентированные на потребителя интернет-провайдеры намеренно не включают многоадресную рассылку в своих маршрутизаторах.

Для отправки и получения многоадресных данных за пределы локальной сети вам необходим многоадресный маршрутизатор. Нужно узнать у администратора, поддерживают ли ваши маршрутизаторы многоадресную рассылку. Можно также попробовать пропинговать `all-routers.mcast.net`. Если какой-либо маршрутизатор отвечает, то ваша сеть подключена к многоадресному маршрутизатору:

`ping all-routers.mcast.net`

```

Администратор: C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.
C:\Users\master>ping all-routers.mcast.net

Обмен пакетами с all-routers.mcast.net [224.0.0.2] с 32 байтами данных:
Превышен интервал ожидания для запроса.
Превышен интервал ожидания для запроса.
Превышен интервал ожидания для запроса.
Превышен интервал ожидания для запроса.

Статистика Ping для 224.0.0.2:
    Пакетов: отправлено = 4, получено = 0, потеряно = 4
    (100% потеря)

C:\Users\master>
  
```

Именно это ограничение может не позволить отправлять или получать данные с каждого многоадресного хоста в *Интернете*. Чтобы пакеты достигли любого удаленного хоста, между вашим хостом и удаленным хостом должен быть путь через многоадресные маршрутизаторы. В качестве альтернативы, некоторые сайты могут быть связаны специальным программным обеспечением для многоадресного туннеля, который передает многоадресные данные по одноадресному *UDP*, который понимают все маршрутизаторы.

## Параметры сокета **DatagramSocket**

Перед тем, как рассмотреть особенности многоадресной передачи данных, обсудим еще два параметра `DatagramSocket`.

### Параметр **SO\_REUSEADDR**

Для *UDP* сокетов параметр `SO_REUSEADDR` означает немного другое, чем одноименный параметр *TCP* сокетов. Для *UDP* сокетов параметр `SO_REUSEADDR` определяет, могут ли несколько датаграммных сокетов связываться с одним и тем же адресом и портом одновременно. Если

несколько датаграмных сокетов связаны с одним и тем же портом, то получаемые на данный порт пакеты будут копироваться во все связанные с данным портом сокет. Этот параметр управляется двумя методами:

```
public void setReuseAddress(boolean on) throws SocketException
public boolean getReuseAddress() throws SocketException
```

По правилам, метод `setReuseAddress()` должен быть вызван до того, как новый сокет будет связан с портом. Это означает, что сокет нужно создать в неподключенном состоянии, с использованием защищенного конструктора, который принимает `DatagramImpl` в качестве аргумента. Следовательно, этот метод не будет работать с обычными `DatagramSocket`. Такие многократно порты чаще всего используются для создания многоадресных сокетов (об этом поговорим на этой лекции чуть позже). Датаграмные каналы также создают неподключенные датаграмные сокет, которые можно настроить для повторного использования портов.

### Параметр **SO\_BROADCAST**

Параметр `SO_BROADCAST` определяет, разрешено ли сокету отправлять пакеты и получать пакеты с широковещательных адресов, таких как `192.168.254.255` – широковещательный адрес для локальной сети с адресом `192.168.254.*`. Широковещательная передача по протоколу *UDP* часто используется для таких протоколов, как, например, *DHCP* (англ. *Dynamic Host Configuration Protocol* – протокол динамической настройки узла), которые должны взаимодействовать с серверами в локальной сети, адреса которых заранее неизвестны. С этим параметром работают два метода:

```
public void setBroadcast(boolean on) throws SocketException
public boolean getBroadcast() throws SocketException
```

Обычно активное сетевое оборудование (такое как, маршрутизаторы и шлюзы) не пересылает широковещательные сообщения, т.к. они могут сильно увеличить объем трафика в локальной сети. По умолчанию этот параметр находится во включенном состоянии. «Отключить» его можно так:

```
socket.setBroadcast(false);
```

Этот параметр может быть изменен после привязки сокета. В некоторых реализациях сокет, привязанные к некоторым определенным адресам, не

принимают широковещательные пакеты. Поэтому при прослушивании широковещательных сообщений рекомендуется использовать конструктор `DatagramPacket(int port)`, а не конструктор `DatagramPacket(InetAddress address, int port)`. Такая процедура необходима в дополнение к установке параметра `SO_BROADCAST` в значение `true`.

### Работа с многоадресными сокетами

Давайте рассмотрим программные средства *Java*, с помощью которых можно организовать многоадресное общение. Для этого предназначены класс `java.net.MulticastSocket`, который является подклассом класса `java.net.DatagramSocket`.

```
public class MulticastSocket
    extends DatagramSocket
    implements Closeable, AutoCloseable
```

Полную информацию по этому классу можно найти на странице с документацией:

[https://docs.oracle.com/javase/8/docs/api/java/net/MulticastSocket.html#MulticastSocket\(int\)](https://docs.oracle.com/javase/8/docs/api/java/net/MulticastSocket.html#MulticastSocket(int))

Мы рассмотрим основные возможности, которые нам понадобятся для работы.

В основном (по свойствам и поведению) объекты класса `MulticastSocket` очень похожи на объекты `DatagramSocket`. Например, схема отправки сообщений остается похожей на ту, что использовалась с обычными датаграммными сокетами. Нужно поместить данные, предназначенные для отправки на другой хост, в объекты типа `DatagramPacket`, которые можно отправлять и получать с помощью объектов типа `MulticastSocket`.

Для того, чтобы получить многоадресное сообщение с какого-либо удаленного сайта, нужно сначала создать объект `MulticastSocket`, который будет прослушивать заданный порт.

```
MulticastSocket ms = new MulticastSocket(2300);
```

Данная строка кода создает объект `MulticastSocket`, который прослушивает порт 2300.

После создания сокета следует присоединиться к многоадресной группе, используя метод `joinGroup()`, характерный для `MulticastSocket`:

```
InetAddress group = InetAddress.getByName("224.2.2.2");
ms.joinGroup(group);
```

Эти действия дают указания маршрутизаторам на пути между вами и сервером, выполнять отправку данных на ваш хост, и сообщает локальному хосту, что он должен программе *IP*-пакеты, адресованные группе многоадресной рассылки.

После того, как выполнено присоединение к группе многоадресной рассылки, программа будет получать данные по *UDP* протоколу в точности так же, как если бы был использован *DatagramSocket*. Нужно создать *DatagramPacket*, содержащий байтовый массив, который будет буфером для данных, и организовать цикл, в котором программа будет получать данные, вызывая метод *receive()*, унаследованный от класса *DatagramSocket*:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
ms.receive(dp);
```

Программа должна прекратить получать многоадресные данные, то нужно покинуть группу многоадресной рассылки, вызвать на многоадресном сокете метод *leaveGroup()*. После этого, можно закрывать сокет с помощью метода *close()*, который унаследован от *DatagramSocket*:

```
ms.leaveGroup(group);
ms.close();
```

Отправка данных на многоадресный адрес аналогична отправке *UDP* данных на одноадресный (*unicast*) хост. Не нужно выполнять операцию присоединения к многоадресной группе для того, чтобы отправить туда данные. Нужно просто создать новый *DatagramPacket*, разместить в нем данные, предназначенные для отправки, и адрес группы многоадресной рассылки, и передать пакет методу *send()*:

```
InetAddress ia = InetAddress.getByName("experiment.mcast.net");
byte[] data = "Here's some multicast data\r\n".getBytes("UTF-8");
int port = 4000;
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
MulticastSocket ms = new MulticastSocket();
ms.send(dp);
```



Есть важное замечание, которое подчеркивают руководства по многоадресной рассылке (*multicast messanging*): многоадресные сокеты создают достаточно большие проблемы с безопасностью. Таким образом, небезопасному коду (*untrusted code*), работающему под управлением менеджера безопасности (*SecurityManager*), не разрешается делать любые действия, затрагивающие многоадресные сокеты. Обычно, коду, загруженному с удаленного хоста, разрешается только отправлять датаграммы на или получать датаграммы с того хоста, с которого они были загружены. Но многоадресные сокеты не позволяют накладывать такого рода ограничения на те пакеты, которые они отправляют или получают. После того, как данные отправлены на многоадресный сокет, очень сложно наложить ограничения на те хосты, которые могут получить эти данные. Поэтому, большинство сред, выполняющих удаленный код обычно запрещают многоадресную рассылку.

## Конструкторы

Класс содержит обычный набор конструкторов. При создании многоадресного сокета можно указать конкретный порт для прослушивания или можно разрешить *Java* назначить анонимный порт:

```
public MulticastSocket() throws SocketException
public MulticastSocket(int port) throws SocketException
public MulticastSocket(SocketAddress bindAddress) throws IOException
```

Например:

```
MulticastSocket ms1 = new MulticastSocket();
MulticastSocket ms2 = new MulticastSocket(4000);
SocketAddress address = new
    InetSocketAddress("192.168.254.32", 4000);
MulticastSocket ms3 = new MulticastSocket(address);
```

Все указанные конструкторы выбрасывают исключение *SocketException*, в случае если *Socket* не может быть создан. Такое обычно случается если недостаточно прав для привязки к указанному при создании порту или порт, к которому создаваемый сокет пытается подключиться, уже используется. Нужно обратить внимание, что с точки зрения операционной системы многоадресный сокет является датаграмным сокетом, то *MulticastSocket* и *DatagramSocket* не могут работать с одним и тем же портом.

Конструктору с параметром можно передать `null`; при этом будет создан несвязанный сокет, который можно будет позже привязать к порту при помощи метода `bind()`. Эта возможность может быть полезна для настройки тех параметров сокета, которые могут быть установлены только до того, как сокет будет привязан. Рассмотрим пример: приведем фрагмент кода, который создает многоадресный сокет с отключенным значением `SO_REUSEADDR` (обычно, по умолчанию, этот параметр включен для многоадресных сокетов):

```
MulticastSocket ms = new MulticastSocket(null);
ms.setReuseAddress(false);
SocketAddress address = new InetSocketAddress(4000);
ms.bind(address);
```

### **Взаимодействие с многоадресной группой**

После того, как объект `MulticastSocket` создан, он может выполнить четыре основные операции:

1. Присоединитесь к многоадресной группе.
2. Отправьте данные членам группы.
3. Получать данные из группы.
4. Выйдите из группы многоадресной рассылки.

Класс `MulticastSocket` содержит специальные методы для операций 1 и 4. Для того, чтобы отправлять или получать данных новые методы не требуются. Для этих операций вполне достаточно методов `send()` и `receive()`, полученных в наследство от суперкласса `DatagramSocket`. Указанные выше операции, в принципе, могут быть выполнены в любом разумно порядке, нужно только помнить, что нужно сначала присоединиться к группе, прежде чем будет можно получать от нее данные. Для того, чтобы отправить данные в группу, нет необходимости к ней присоединяться; кроме того, можно свободно чередовать отправку и получение данных.

### **Присоединение к группам многоадресной рассылки**

Для того, чтобы присоединиться к группе многоадресной рассылки нужно передать объект или типа `InetAddress`, или типа `SocketAddress`, представляющий группу многоадресной рассылки, методу `joinGroup()`:

```
public void joinGroup(InetAddress address) throws IOException
public void joinGroup(SocketAddress address,
                     NetworkInterface interface) throws IOException
```

После того, как было успешно выполнено присоединение к многоадресной группе, можно получать датаграммы точно так же, как в случае одноадресных датаграмм: передаем объект `DatagramPacket`, созданный для приема датаграм, в метод `receive()` этого сокета. Например:

```
try {
    MulticastSocket ms = new MulticastSocket(4000);
    InetAddress ia = InetAddress.getByName("224.2.2.2");
    ms.joinGroup(ia);
    byte[] buffer = new byte[8192];
    while (true) {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        ms.receive(dp);
        String s = new String(dp.getData(), "8859_1");
        System.out.println(s);
    }
} catch (IOException ex) {
    System.err.println(ex);
}
```

Если осуществляется попытка присоединения не к многоадресному (*multicast*) адресу (если он находится не между адресами 224.0.0.0 и 239.255.255.255), метод `joinGroup()` выбросит `IOException`.

Один объект `MulticastSocket` может быть членом нескольких групп многоадресной рассылки. Информация о членстве в многоадресных группах хранится в многоадресных маршрутизаторах, а не в объекте. В этом случае для того, чтобы определить, адрес, с которого пришем пакет, нужно получить адрес, сохраненный во входящей дейтаграмме.

На одном компьютере, и даже в одной и той же программе *Java*, несколько сокетов многоадресной рассылки могут присоединиться к одной группе. В этом случае, каждый сокет получает полную копию данных, адресованных этой группе, которые поступают на локальный хост.

Второй аргумент метода дает возможность присоединиться к многоадресной группе только на указанном локальном сетевом интерфейсе. Рассмотрим пример, в котором выполняется попытка присоединиться к многоадресной группе с *IP*-адресом 224.2.2.2 на сетевом интерфейсе с именем «eth0», если такой интерфейс существует. Если такого интерфейса не

существует, то будет попытка присоединения ко всем доступным сетевым интерфейсам:

```
MulticastSocket ms = new MulticastSocket();
SocketAddress group = new InetSocketAddress("224.2.2.2", 40);
NetworkInterface ni = NetworkInterface.getByName("eth0");
if (ni != null) {
    ms.joinGroup(group, ni);
} else {
    ms.joinGroup(group);
}
```

Данный метод работает почти так же, как и метод `joinGroup()` с одним аргументом. Например, передача в качестве первого аргумента объекта `SocketAddress`, который не представляет многоадресную группу, вызовет исключение `IOException`.

### **Выход из группы и закрытие соединения**

Для того, чтобы покинуть группу и перестать получать из нее датаграммы многоадресной рассылки, либо на всех, либо на указанном сетевом интерфейсе нужно вызвать метод `leaveGroup()`:

```
public void leaveGroup(InetAddress address) throws IOException
public void leaveGroup(SocketAddress multicastAddress,
                       NetworkInterface interface) throws IOException
```

Вызов метода передает сигнал локальному многоадресному маршрутизатору, чтобы он прекратил отправлять датаграммы на сокет. Если адрес, который указан в качестве аргумента метода, не является многоадресным (если он не между 224.0.0.0 и 239.255.255.255), то метод выбросит исключение `IOException`. При этом разрешается покидать многоадресную группу, к которой не присоединялись.

Практически все методы `MulticastSocket` могут выбрасывать исключение `IOException`; обычно вся работа выполняется в блоке `try`. В современной *Java* класс `DatagramSocket` имплементирует интерфейс `Autocloseable`, таким образом можно использовать `try-c-ресурсами`:

```
try (MulticastSocket socket = new MulticastSocket()) {
```

```

    // connect to the server...
} catch (IOException ex) {
    ex.printStackTrace();
}

```

В ранних версиях *Java* (с *Java* 6 и более ранних) для освобождения ресурсов нужно было явно закрыть сокет в блоке `finally`:

```

MulticastSocket socket = null;
try {
    socket = new MulticastSocket();
    // connect to the server...
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    if (socket != null) {
        try {
            socket.close();
        } catch (IOException ex) {
            // ignore
        }
    }
}

```

### Отправка многоадресных сообщений

Отправка данных с помощью объекта `MulticastSocket` выполняется аналогично отправке данных с помощью `DatagramSocket`. Поместите информацию, которую нужно отправить, в объект `DatagramPacket` и отправьте его с методом `send()`, который унаследован от класса `DatagramSocket`. Данные отправляются каждому хосту, который принадлежит группе многоадресной рассылки, в которую направляется пакет. Например:

```

try {
    InetAddress ia = InetAddress.getByName("experiment.mcast.net");
    byte[] data = "Here's some multicast data\r\n".getBytes();
    int port = 4000;
    DatagramPacket dp = new DatagramPacket(data, data.length,

```

```

                                ia, port);
    MulticastSocket ms = new MulticastSocket();
    ms.send(dp);
} catch (IOException ex) {
    System.err.println(ex);
}

```

По умолчанию в многоадресных сокетах используется параметр *TTL* (*Time to live*), равный 1 (то есть пакеты не выходят за пределы локальной подсети). Однако, в случае необходимости, можно изменить этот параметр для отдельного пакета, передав целое число от 0 до 255 в качестве второго параметра методу отправки, переопределенному в классе `MulticastSocket` (метод `send(DatagramPacket dp, byte ttl)`).

Кроме того, можно воспользоваться методом `setTimeToLive()` класса `MulticastSocket`, который устанавливает значение *TTL* по умолчанию, используемое для пакетов, отправляемых из сокета с использованием метода `send(DatagramPacket dp)`, унаследованного от `DatagramSocket`. Для того, чтобы узнать текущее значение параметра, можно вызвать метод `getTimeToLive()`, который возвращает значение *TTL* по умолчанию для данного объекта `MulticastSocket`:

```

public void setTimeToLive(int ttl) throws IOException
public int getTimeToLive() throws IOException

```

Например, приведем фрагмент кода, который устанавливает *TTL* равным 64:

```

try {
    InetAddress ia = InetAddress.getByName("experiment.mcast.net");
    byte[] data = "Here's some multicast data\r\n".getBytes();
    int port = 4000;
    DatagramPacket dp = new DatagramPacket(data, data.length,
                                            ia, port);

    MulticastSocket ms = new MulticastSocket();
    ms.setTimeToLive(64);
    ms.send(dp);
} catch (IOException ex) {
    System.err.println(ex);
}

```



}

### Режим обратной петли (*loopback*)

При работе в многоадресном режиме возникает проблема – должен ли хост принимать многоадресные пакеты, которые он сам же отправляет. Вообще-то возвращаются ли они обратно или нет, зависит от платформы. Передача `true` в метод `setLoopback()` означает, что хост не хочет получать отправленные им пакеты. Передача `false` этому методу означает, что хост действительно хочет получать отправленные им пакеты:

```
public void setLoopbackMode(boolean disable) throws SocketException
public boolean getLoopbackMode() throws SocketException
```

Однако это только подсказка для нативного (платформенно-ориентированного) сетевого кода платформы и реализации не обязаны делать это. Поскольку режим обратной петли в системе может не поддерживаться, то в случае если пакеты и отправляются, и получаются, бывает нужно проверить, поддерживается ли системой режим обратной петли (*loopback*). Метод `getLoopbackMode()` возвращает `true`, если пакеты не возвращаются (*not looped back*), и `false`, если возвращаются. (Этот метод, скорее всего, был написан в соответствии с соглашением, что значения по умолчанию всегда должны давать `true`).

Но, вообще-то это поведение зависит от системы. Можно запросить желаемое поведение с помощью метода `setLoopback()`, но на это нельзя рассчитывать. Если система принимает отосланные пакеты (*looping packets back*), а этого не требуется и изменить это поведение не получается, то нужно каким-то образом пометить пакеты и отбрасывать их сразу после приема. Если система не принимает обратно отосланные пакеты, а это нужно и изменить это поведение не получается, то нужно сохранять копии отправляемых пакетов и размещать их нужные внутренние структуры данных одновременно с отправкой.

### Сетевые интерфейсы

На хосте с несколькими сетевыми интерфейсами (*multihomed host*) методы `setInterface()` и `setNetworkInterface()` указывают сетевой интерфейс, используемый для многоадресной отправки и получения:

```
public void setInterface(InetAddress address) throws SocketException
public InetAddress getInterface() throws SocketException
```

```
public void setNetworkInterface(NetworkInterface interface)
                                   throws SocketException
public NetworkInterface getNetworkInterface() throws SocketException
```

Указанные методы установки вызывают исключение `SocketException` в тех случаях, когда аргумент не является адресом сетевого интерфейса на конкретной локальной машине. Заметим, что сетевой интерфейс всегда неизменяем (*immutablely*) и устанавливается в конструкторе для одноадресных (*unicast*) объектов `Socket` и `DatagramSocket`, но является изменяемым и задается отдельным методом для объектов `MulticastSocket`. Рекомендуется, для безопасности, установить интерфейс сразу после создания `MulticastSocket` и после этого его не менять. Пример использования метода `setInterface()`:

```
try {
    InetAddress ia = InetAddress.getByName("www.ibiblio.org");
    MulticastSocket ms = new MulticastSocket(2048);
    ms.setInterface(ia);
    // send and receive data...
} catch (UnknownHostException ue) {
    System.err.println(ue);
} catch (SocketException se) {
    System.err.println(se);
}
```

Метод `setNetworkInterface()` предназначен для тех же целей, что и метод `setInterface()`; то есть он указывает сетевой интерфейс, который будет использован для многоадресной отправки и получения. Отличие – в параметрах: в первом случае указывается имя локального сетевого интерфейса, такого как например «eth0» (инкапсулирован в объекте `NetworkInterface`), а во втором – *IP*-адрес, связанный с этим сетевым интерфейсом (инкапсулирован в объекте `InetAddress`). Метод `setNetworkInterface()` может выбросить исключение `SocketException`, если объект `NetworkInterface`, переданный в качестве аргумента, не является сетевым интерфейсом локальной машины.

Метод `getNetworkInterface()` возвращает объект `NetworkInterface`, представляющий сетевой интерфейс, на котором анализируемый объект `MulticastSocket` прослушивает данные. Если сетевой интерфейс не был явно

указан в конструкторе или с помощью метода `setNetworkInterface()`, то будет возвращает объект-заполнитель с адресом «0.0.0.0» и номером порта –1. Рассмотрим фрагмент кода, который выводит сетевой интерфейс, используемый сокетом:

```
NetworkInterface intf = ms.getNetworkInterface();
System.out.println(intf.getName());
```

## Примеры

Рассмотрим несколько примеров простых демонстрационных приложений.

### Простой полный пример

Рассмотрим примеры простых демонстрационных сетевых приложений, реализованных с помощью многоадресных сокетов.

Сначала напишем приложение для демонстрации группового общения. Одна часть приложения отправляет строку в многоадресную группу, а некоторое количество хостов – приемников получают эту строку из многоадресной группы. Обе части приложения (и та, что отправляет строку, и та, что принимает строку) созданы с учетом схемы построения сетевых приложений на *UDP* сокетах.

#### *Часть приложения, которая отправляет данные*

```
import java.net.*;

public class HelloSender {
    public static void main(String[] args) {
        try {
            // Create a Datagram Socket
            DatagramSocket socket = new DatagramSocket();
            // Fill the Buffer with Data
            String msg = "Hello";
            byte[] out = msg.getBytes();
            // Multicast group where packet has to sent
            InetAddress group = InetAddress.getByName("224.0.0.1");
            //Port the receiver listens on
            int port = 8379;
            //Create the DatagramPacket with buffer, address and port
            DatagramPacket packet = new DatagramPacket(out, out.length, group, port);
            //Send to multicast IP address and port
            System.out.println("Sending a packet...");
            //Send the packet now
            socket.send(packet);
            System.out.println("Sent : " + msg);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

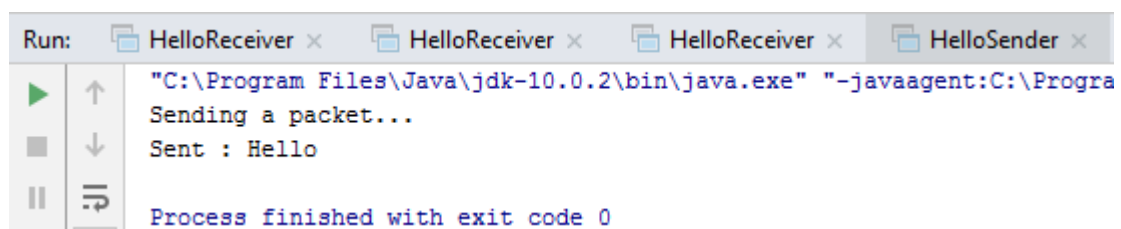
Следует обратить внимание на то, что для отправки *UDP* сообщений в многоадресную группу можно использовать обычный датаграмный сокет.

### *Часть приложения, которая отправляет данные*

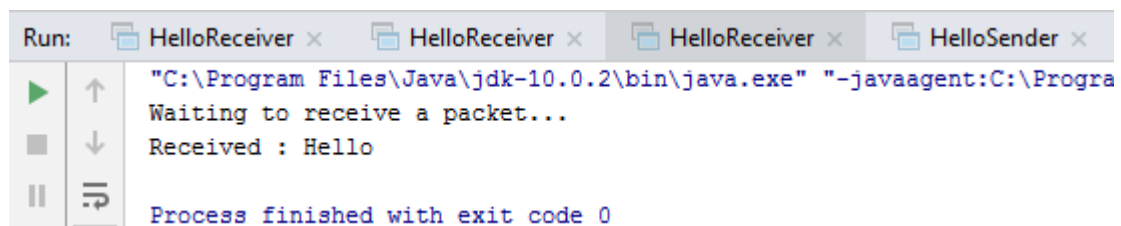
```
import java.net.*;

public class HelloReceiver {
    public static void main(String[] args) {
        try {
            //Create a MulticastSocket and bind it to port 8379
            MulticastSocket socket = new MulticastSocket(8379);
            //Join to multicast group
            socket.joinGroup(InetAddress.getByName("224.0.0.1"));
            //Construct a DatagramPacket to receive packet
            byte[] in = new byte[256];
            DatagramPacket packet = new DatagramPacket(in, in.length);
            System.out.println("Waiting to receive a packet...");
            //Receive the packet now and display
            socket.receive(packet);
            String msg = new String(in, 0, packet.getLength());
            System.out.println("Received : " + msg);
        } catch (Exception ioe) {
            System.out.println(ioe);
        }
    }
}
```

Стандартным образом запускаем приложение. Запустим экземпляра три приемника сообщения и один экземпляр отправителя сообщения. Приведем рисунки, демонстрирующие работу приложения. Часть, отправляющая сообщение в многоадресную группу:



Часть приложения, принимающая сообщение:



Аналогичные сообщения будут «одновременно» выведено в каждом из трех запущенных примеников сообщений из многоадресной группы.

## Второй пример

Рассмотрим еще один простой демонстрационный пример, в котором одна часть приложения постоянно передает искусственно созданный счет некоего спортивного матча в многоадресную группу. Любой клиент (вторая часть приложения), которой захочет узнать текущий счет, может в произвольное время присоединиться к группе и отслеживать счет матча. При желании, можно отсоединиться от группы, и присоединиться в любой момент позже.

### Отправляющая часть приложения

```
import java.io.*;
import java.net.*;
import java.util.Random;

public class ScoreSender {
    public static void main(String[] args) {
        long score = 0, run;
        Random r = new Random();
        try {
            int port = 8379;
            //InetAddress group = InetAddress.getByName(args[0]);
            InetAddress group = InetAddress.getByName("224.0.0.1");
            //Create a DatagramSocket
            DatagramSocket socket = new DatagramSocket();
            while(true) {
                //Fill the buffer with score generated artificially
                do {
                    Thread.sleep(1000+r.nextInt(1000));
                } while((run = r.nextInt(7)) == 0);
                score += run;
                String msg = "score: " + score;
                byte[] out = msg.getBytes();
                //Create a DatagramPacket
                DatagramPacket pkt = new DatagramPacket(out, out.length, group, port);
                //Send the pkt
                socket.send(pkt);
                System.out.println("Send-->" + msg);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

### Принимающая часть приложения

```
import java.io.*;
import java.net.*;

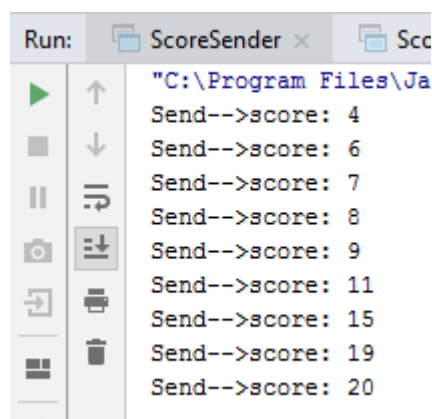
public class ScoreReceiver {
```

```

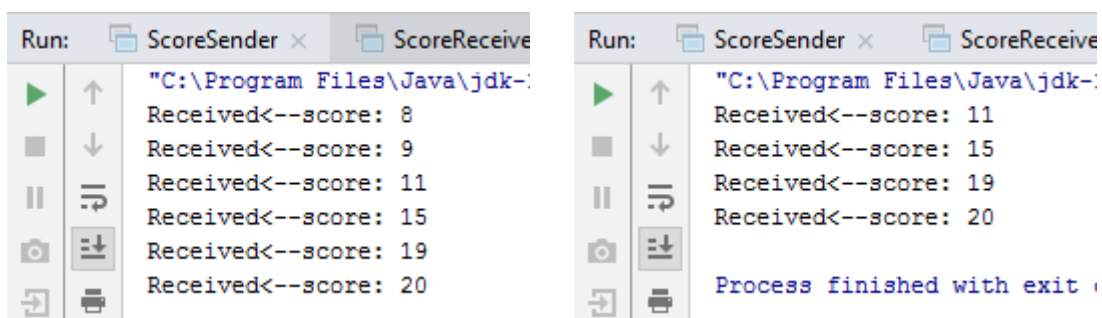
public static void main(String[] args) {
    byte[] inBuffer = new byte[256];
    try {
        InetAddress address = InetAddress.getByName("224.0.0.1");
        //Create a MulticastSocket
        MulticastSocket socket = new MulticastSocket(8379);
        //Join to the multicast group
        socket.joinGroup(address);
        while (true) {
            DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
            socket.receive(packet);
            String msg = new String(inBuffer, 0, packet.getLength());
            System.out.println("Received<--" + msg);
        }
    } catch (IOException ioe) {
        System.out.println(ioe);
    }
}
}

```

Стандартным образом запускаем приложение. Часть приложения, искусственно генерирующая счет и отправляющая его в многоадресную группу:



Части, присоединившиеся к многоадресной группе и принимающие сообщения, которые были в группу направлены:



Приведены снимки экранов двух получателей сообщений. Можно видеть, что они в разное время присоединились к группе.



## Третий пример

Немного изменим предыдущий пример. Сделаем две группы: в одну будет направляться счет, а во вторую – случайным образом сгенерированные даты. Один клиент-приемник сообщений будет получать информацию из двух групп сразу, а один – только из одной. Приведем полный код приложения.

### Отправляющая часть приложения

```
import java.io.*;
import java.net.*;
import java.util.Random;

public class ScoreSender {
    public static void main(String[] args) {
        long score = 0, run;
        Random r = new Random();
        try {
            int port = 8379;
            //InetAddress group = InetAddress.getByName(args[0]);
            InetAddress group = InetAddress.getByName("224.0.0.1");
            //Create a DatagramSocket
            DatagramSocket socket = new DatagramSocket();
            while(true) {
                //Fill the buffer with score generated artificially
                do {
                    Thread.sleep(1000+r.nextInt(1000));
                } while((run = r.nextInt(7)) == 0);
                score += run;
                String msg = "score: " + score;
                byte[] out = msg.getBytes();
                //Create a DatagramPacket
                DatagramPacket pkt = new DatagramPacket(out, out.length, group, port);
                //Send the pkt
                socket.send(pkt);
                System.out.println("Send-->" + msg);
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.*;
import java.net.*;
import java.time.YearMonth;
import java.util.Random;
import java.util.concurrent.ThreadLocalRandom;

public class DateSender {
    public static void main(String[] args) {
        long run;
        Random r = new Random();
        try {
            int port = 8379;
            //InetAddress group = InetAddress.getByName(args[0]);
            InetAddress group = InetAddress.getByName("224.0.0.2");
            //Create a DatagramSocket
            DatagramSocket socket = new DatagramSocket();
```

```

while(true) {
    //Fill the buffer with score generated artificially
    do {
        Thread.sleep(1000+r.nextInt(1000));
    } while((run = r.nextInt(7)) == 0);
    //Получаем текущий год + месяц
    YearMonth now = YearMonth.now();
    //Генерируем случайное число от 0 до 36
    int randomNumber = ThreadLocalRandom.current().nextInt(37);
    YearMonth randomDate = now.plusMonths(randomNumber);
    String msg = "date: " + randomDate.getMonthValue() + "." +
        randomDate.getYear();
    byte[] out = msg.getBytes();
    //Create a DatagramPacket
    DatagramPacket pkt = new DatagramPacket(out, out.length, group, port);
    //Send the pkt
    socket.send(pkt);
    System.out.println("Send-->" + msg);
}
} catch(Exception e) {
    e.printStackTrace();
}
}
}

```

## Принимающая часть приложения

```

import java.io.*;
import java.net.*;

public class ScoreReceiver {
    public static void main(String[] args) {
        byte[] inBuffer = new byte[256];
        try {
            InetAddress address = InetAddress.getByName("224.0.0.1");
            //Create a MulticastSocket
            MulticastSocket socket = new MulticastSocket(8379);
            //Join to the multicast group
            socket.joinGroup(address);
            while (true) {
                DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
                socket.receive(packet);
                String msg = new String(inBuffer, 0, packet.getLength());
                System.out.println("Received<--" + msg);
            }
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}

```

```

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class DataReceiver {
    public static void main(String[] args) {
        byte[] inBuffer = new byte[256];
        try {
            InetAddress address = InetAddress.getByName("224.0.0.2");

```

```

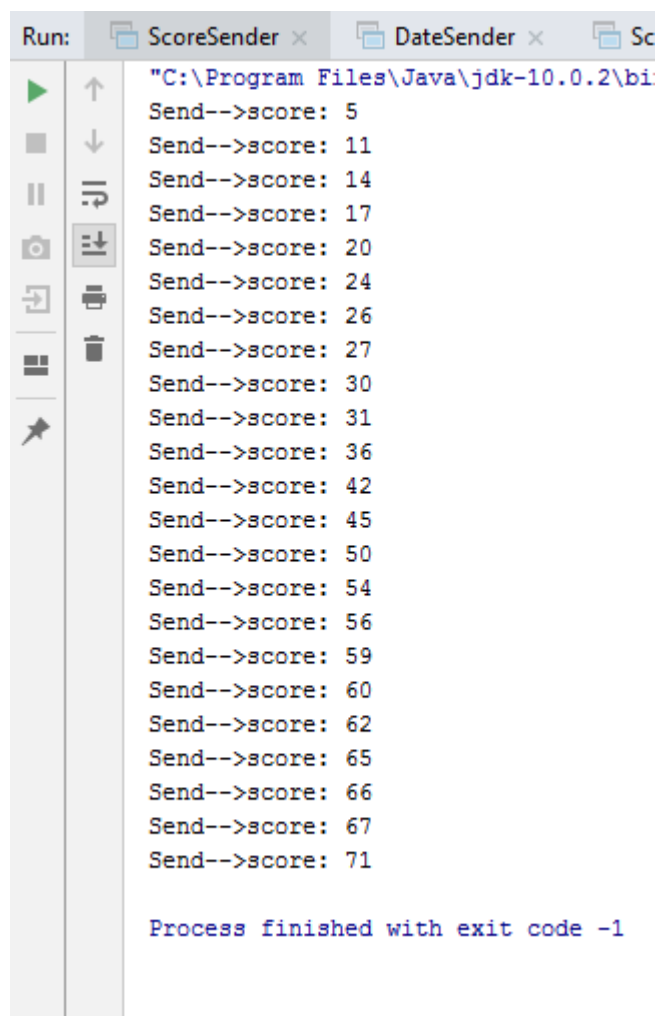
//Create a MulticastSocket
MulticastSocket socket = new MulticastSocket(8379);
//Join to the multicast groups
socket.joinGroup(address);
while (true) {
    DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
    socket.receive(packet);
    String msg = new String(inBuffer, 0, packet.getLength());
    System.out.println("Received<--" + msg);
}
} catch (IOException ioe) {
    System.out.println(ioe);
}
}
}

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class ScoreDataReceiver {
    public static void main(String[] args) {
        byte[] inBuffer = new byte[256];
        try {
            InetAddress address1 = InetAddress.getByName("224.0.0.1");
            InetAddress address2 = InetAddress.getByName("224.0.0.2");
            //Create a MulticastSocket
            MulticastSocket socket = new MulticastSocket(8379);
            //Join to the multicast groups
            socket.joinGroup(address1);
            socket.joinGroup(address2);
            while (true) {
                DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
                socket.receive(packet);
                String msg = new String(inBuffer, 0, packet.getLength());
                System.out.println("Received<--" + msg + " From: " +
                                   packet.getSocketAddress());
            }
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
}

```

Стандартным образом запустим приложение и приведем снимки экрана. Часть приложения, отправляющая данные в многоадресную группу:

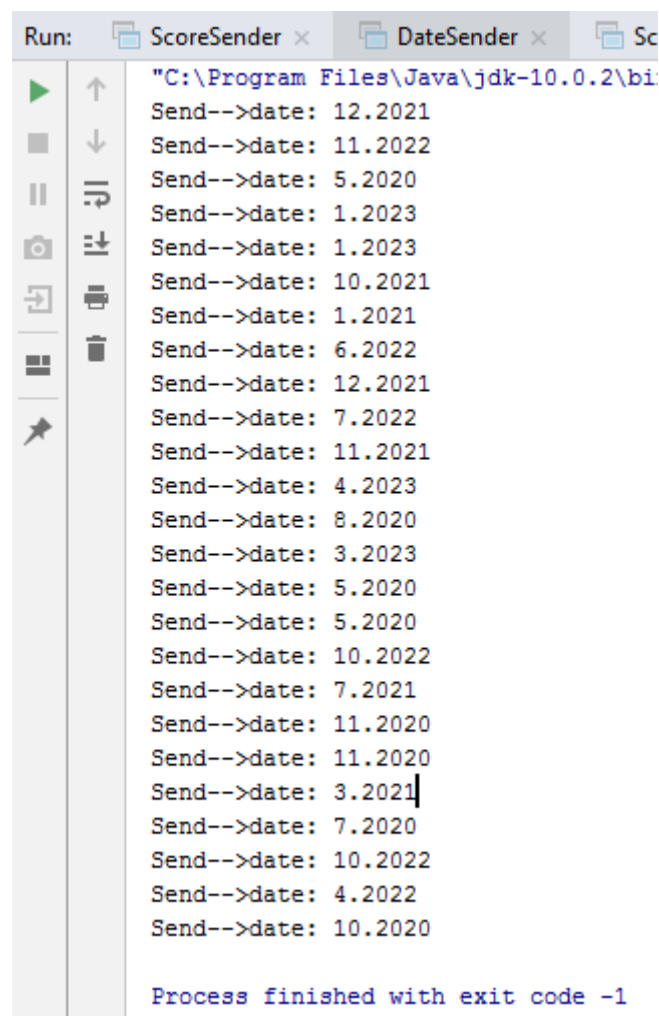


```

Run: ScoreSender x DateSender x Sc
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Send-->score: 5
Send-->score: 11
Send-->score: 14
Send-->score: 17
Send-->score: 20
Send-->score: 24
Send-->score: 26
Send-->score: 27
Send-->score: 30
Send-->score: 31
Send-->score: 36
Send-->score: 42
Send-->score: 45
Send-->score: 50
Send-->score: 54
Send-->score: 56
Send-->score: 59
Send-->score: 60
Send-->score: 62
Send-->score: 65
Send-->score: 66
Send-->score: 67
Send-->score: 71

Process finished with exit code -1

```



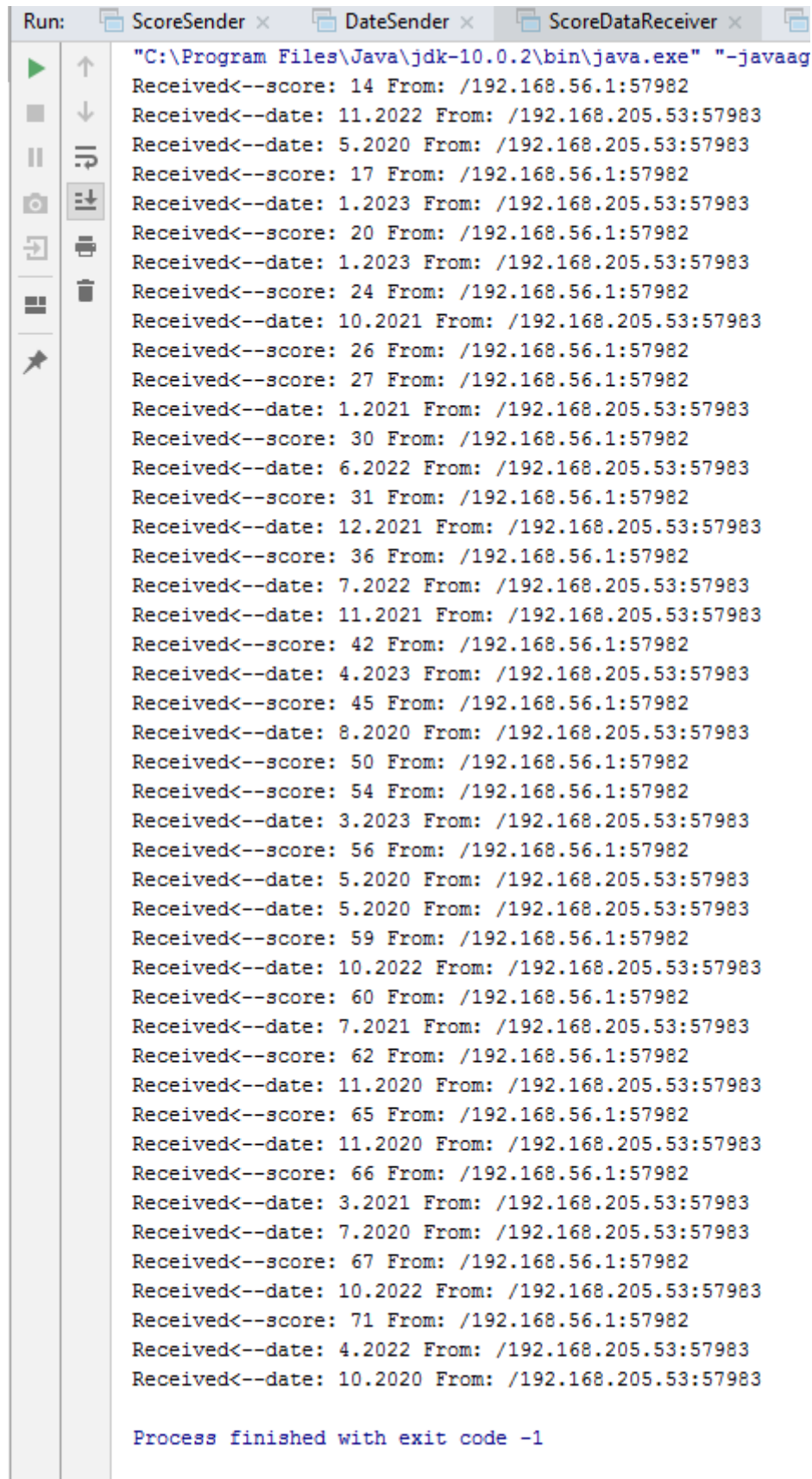
```

Run: ScoreSender x DateSender x Sc
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Send-->date: 12.2021
Send-->date: 11.2022
Send-->date: 5.2020
Send-->date: 1.2023
Send-->date: 1.2023
Send-->date: 10.2021
Send-->date: 1.2021
Send-->date: 6.2022
Send-->date: 12.2021
Send-->date: 7.2022
Send-->date: 11.2021
Send-->date: 4.2023
Send-->date: 8.2020
Send-->date: 3.2023
Send-->date: 5.2020
Send-->date: 5.2020
Send-->date: 10.2022
Send-->date: 7.2021
Send-->date: 11.2020
Send-->date: 11.2020
Send-->date: 3.2021
Send-->date: 7.2020
Send-->date: 10.2022
Send-->date: 4.2022
Send-->date: 10.2020

Process finished with exit code -1

```

Приведем снимки экранов принимающих частей приложения. Клиент, состоящий в двух многоадресных группах:



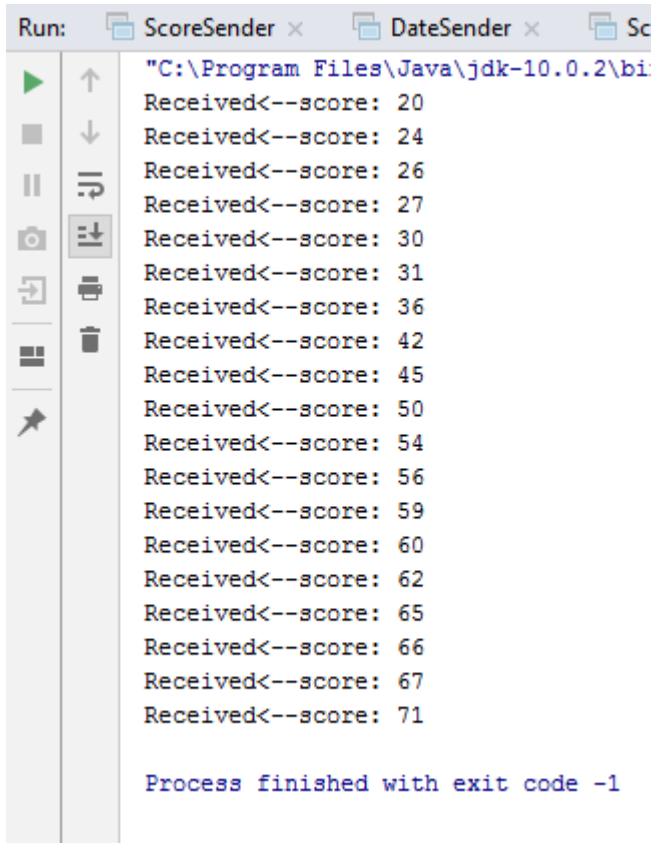
```

Run: ScoreSender x DateSender x ScoreDataReceiver x
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe" "-javaag
Received<--score: 14 From: /192.168.56.1:57982
Received<--date: 11.2022 From: /192.168.205.53:57983
Received<--date: 5.2020 From: /192.168.205.53:57983
Received<--score: 17 From: /192.168.56.1:57982
Received<--date: 1.2023 From: /192.168.205.53:57983
Received<--score: 20 From: /192.168.56.1:57982
Received<--date: 1.2023 From: /192.168.205.53:57983
Received<--score: 24 From: /192.168.56.1:57982
Received<--date: 10.2021 From: /192.168.205.53:57983
Received<--score: 26 From: /192.168.56.1:57982
Received<--score: 27 From: /192.168.56.1:57982
Received<--date: 1.2021 From: /192.168.205.53:57983
Received<--score: 30 From: /192.168.56.1:57982
Received<--date: 6.2022 From: /192.168.205.53:57983
Received<--score: 31 From: /192.168.56.1:57982
Received<--date: 12.2021 From: /192.168.205.53:57983
Received<--score: 36 From: /192.168.56.1:57982
Received<--date: 7.2022 From: /192.168.205.53:57983
Received<--date: 11.2021 From: /192.168.205.53:57983
Received<--score: 42 From: /192.168.56.1:57982
Received<--date: 4.2023 From: /192.168.205.53:57983
Received<--score: 45 From: /192.168.56.1:57982
Received<--date: 8.2020 From: /192.168.205.53:57983
Received<--score: 50 From: /192.168.56.1:57982
Received<--score: 54 From: /192.168.56.1:57982
Received<--date: 3.2023 From: /192.168.205.53:57983
Received<--score: 56 From: /192.168.56.1:57982
Received<--date: 5.2020 From: /192.168.205.53:57983
Received<--date: 5.2020 From: /192.168.205.53:57983
Received<--score: 59 From: /192.168.56.1:57982
Received<--date: 10.2022 From: /192.168.205.53:57983
Received<--score: 60 From: /192.168.56.1:57982
Received<--date: 7.2021 From: /192.168.205.53:57983
Received<--score: 62 From: /192.168.56.1:57982
Received<--date: 11.2020 From: /192.168.205.53:57983
Received<--score: 65 From: /192.168.56.1:57982
Received<--date: 11.2020 From: /192.168.205.53:57983
Received<--score: 66 From: /192.168.56.1:57982
Received<--date: 3.2021 From: /192.168.205.53:57983
Received<--date: 7.2020 From: /192.168.205.53:57983
Received<--score: 67 From: /192.168.56.1:57982
Received<--date: 10.2022 From: /192.168.205.53:57983
Received<--score: 71 From: /192.168.56.1:57982
Received<--date: 4.2022 From: /192.168.205.53:57983
Received<--date: 10.2020 From: /192.168.205.53:57983

Process finished with exit code -1

```

Клиенты, принимающие информацию из одной многоадресной группы.

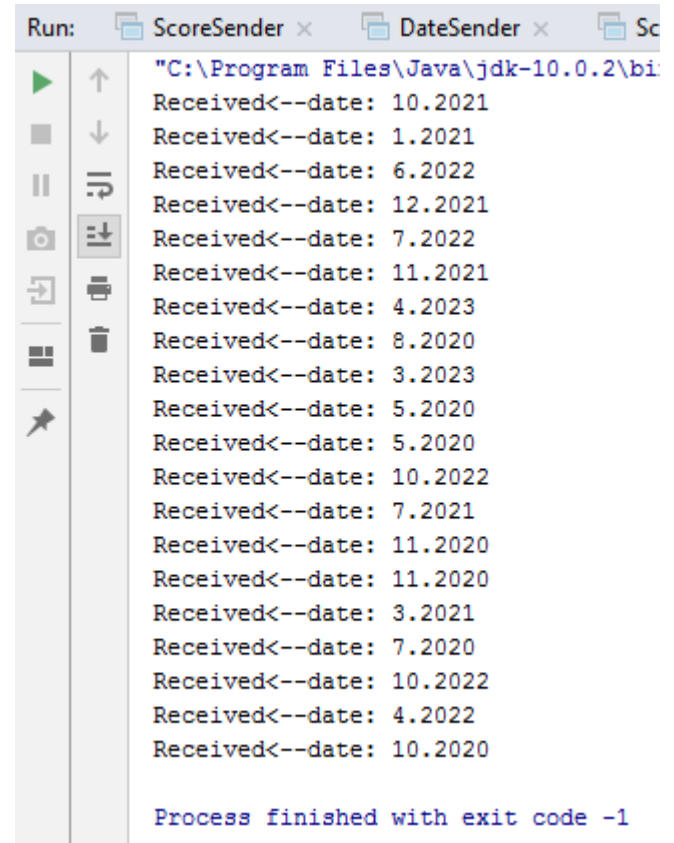


```

Run: ScoreSender x DateSender x Sc
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Received<--score: 20
Received<--score: 24
Received<--score: 26
Received<--score: 27
Received<--score: 30
Received<--score: 31
Received<--score: 36
Received<--score: 42
Received<--score: 45
Received<--score: 50
Received<--score: 54
Received<--score: 56
Received<--score: 59
Received<--score: 60
Received<--score: 62
Received<--score: 65
Received<--score: 66
Received<--score: 67
Received<--score: 71

Process finished with exit code -1

```



```

Run: ScoreSender x DateSender x Sc
"C:\Program Files\Java\jdk-10.0.2\bin\java.exe"
Received<--date: 10.2021
Received<--date: 1.2021
Received<--date: 6.2022
Received<--date: 12.2021
Received<--date: 7.2022
Received<--date: 11.2021
Received<--date: 4.2023
Received<--date: 8.2020
Received<--date: 3.2023
Received<--date: 5.2020
Received<--date: 5.2020
Received<--date: 10.2022
Received<--date: 7.2021
Received<--date: 11.2020
Received<--date: 11.2020
Received<--date: 3.2021
Received<--date: 7.2020
Received<--date: 10.2022
Received<--date: 4.2022
Received<--date: 10.2020

Process finished with exit code -1

```

## Еще два примера

Большинство многоадресных серверов отправляют сообщения всем клиентам, которые к ним подключились. Поэтому можно присоединиться к какой-либо многоадресной группе и просматривать данные, которые ей отправляются. Рассмотрим пример - это класс MulticastSniffer, который пытается присоединиться к группе многоадресной рассылки и получить направляющиеся на нее дейтаграммы.

```

package multicast;

import java.io.*;
import java.net.*;

public class MulticastSniffer {
    public static void main(String[] args) {
        InetAddress group = null;
        int port = 0;
        // read the address from the command line
        try {
            //group = InetAddress.getByName(args[0]);
            group = InetAddress.getByName("239.255.255.250");
            //group = InetAddress.getByName("all-systems.mcast.net");
            //port = Integer.parseInt(args[1]);
            port = Integer.parseInt("1900");
            //port = Integer.parseInt("4000");
        } catch (ArrayIndexOutOfBoundsException | NumberFormatException
            | UnknownHostException ex) {
            System.err.println(
                "Usage: java MulticastSniffer multicast_address port");
            System.exit(1);
        }
    }
}

```



```

MulticastSocket ms = null;
try {
    ms = new MulticastSocket(port);
    ms.joinGroup(group);
    byte[] buffer = new byte[8192];
    while (true) {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        ms.receive(dp);
        String s = new String(dp.getData(), "8859_1");
        System.out.println(s);
    }
} catch (IOException ex) {
    System.err.println(ex);
} finally {
    if (ms != null) {
        try {
            ms.leaveGroup(group);
            ms.close();
        } catch (IOException ex) {}
    }
}

}

/*
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1

M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1

M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1

M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
Man:"ssdp:discover"
MX:3

M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
Man:"ssdp:discover"
MX:3

M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
Man:"ssdp:discover"

```

MX:3

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1
```

MX:3

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 1
ST: urn:dial-multiscreen-org:service:dial:1
```

MX:3

```
M-SEARCH * HTTP/1.1
Host:239.255.255.250:1900
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
Man:"ssdp:discover"
MX:3
*/
```

Посмотрите, как работает приложение (в принципе, оно может ничего и не вывести на экран). Когда устройство *Universal Plug and Play (UPnP)* подключается к сети, оно отправляет сообщение *HTTPU* (*HTTP* через *UDP*) на адрес многоадресной рассылки 239.255.255.250 на порт 1900. Можно попробовать с помощью этой программы прослушать эти сообщения. Если такое устройство работает в сети, программа должна вывести сообщения. В виде комментария приведено несколько сообщений. Похоже на то, что это мой домашний телевизор (с поддержкой *Smart TV*) подключился к локальной сети и работает, регулярно отправляя сообщения. Большинство устройств выводят сообщения только тогда, когда они впервые подключились к сети или когда их запрашивает другое устройство.

Еще раз рассмотрим отправку многоадресных данных.

```
import java.io.*;
import java.net.*;
public class MulticastSender {
    public static void main(String[] args) {
        InetAddress ia = null;
        int port = 0;
        byte ttl = (byte) 1;
        // read the address from the command line
        try {
            //ia = InetAddress.getByName(args[0]);
            ia = InetAddress.getByName("all-systems.mcast.net");
            //port = Integer.parseInt(args[1]);
            port = Integer.parseInt("4000");
            if (args.length > 2) ttl = (byte) Integer.parseInt(args[2]);
        } catch (NumberFormatException | IndexOutOfBoundsException
```

```

        | UnknownHostException ex) {
    System.err.println(ex);
    System.err.println(
        "Usage: java MulticastSender multicast_address port ttl");
    System.exit(1);
}

byte[] data = "Here's some multicast data".getBytes();
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);

try (MulticastSocket ms = new MulticastSocket()) {
    ms.setTimeToLive(ttl);
    ms.joinGroup(ia);
    for (int i = 1; i < 10; i++) {
        ms.send(dp);
    }
    ms.leaveGroup(ia);
} catch (SocketException ex) {
    System.err.println(ex);
} catch (IOException ex) {
    System.err.println(ex);
}
}
}

```

В предыдущем классе – приемнике многоадресном сообщений (MulticastSniffer) изменим многоадресную группу и номер порта.

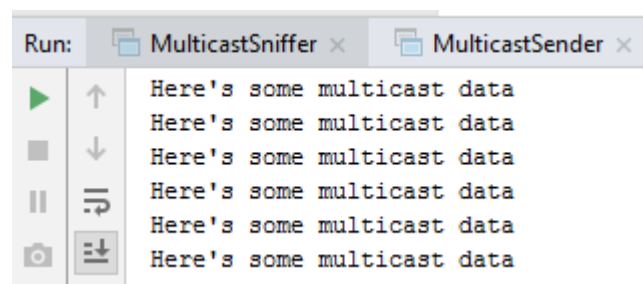
```

//group = InetAddress.getBy_name(args[0]);
//group = InetAddress.getBy_name("239.255.255.250");
group = InetAddress.getBy_name("all-systems.mcast.net");
//port = Integer.parseInt(args[1]);
//port = Integer.parseInt("1900");
port = Integer.parseInt("4000");

```

После этого запустим MulticastSniffer и будем прослушивать многоадресную группу all-systems.mcast.net на порту 4000. Затем запустим MulticastSender на другом компьютере в вашей локальной подсети (или в другом окне на том же компьютере).

Сниффер выведет сообщения:



Чтобы этот пример работал и за пределами локальной подсети, каждая из двух подсетей должна иметь многоадресные маршрутизаторы, а между маршрутизаторами должна быть включена многоадресная рассылка.