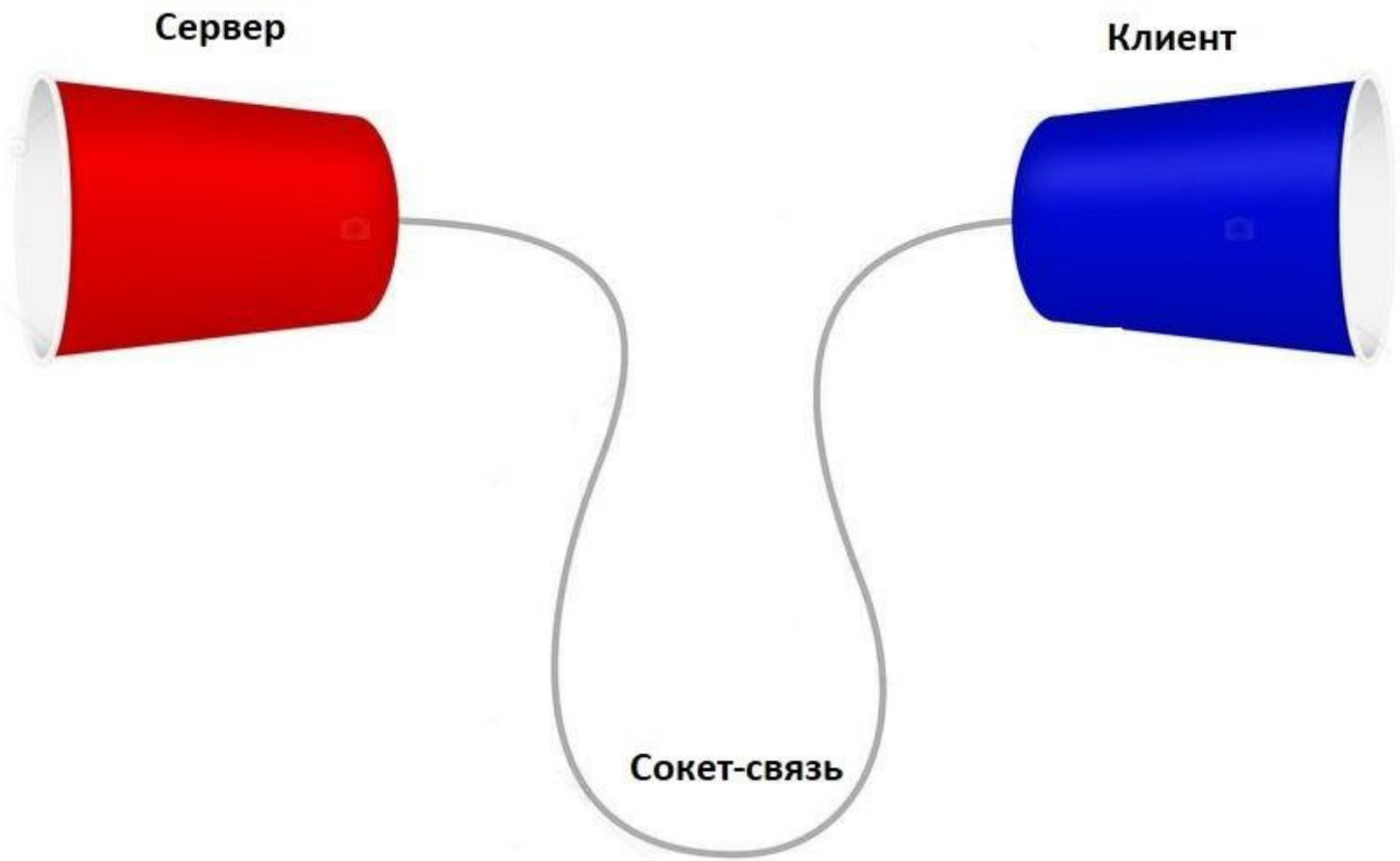
1 Соединение

Для объяснения как же происходит сетевое соединение между сервером и одним клиентом, возьмем, ставший уже классическим, пример с многоквартирным домом. Допустим, клиенту нужно каким-то образом установить связь с определённым сервером. Что нужно знать ищущему об объекте поиска? Да, адрес. Сервер, это не магическая сущность на облаке, и поэтому он должен находиться на определённой машине. По аналогии с домом, где должна произойти встреча двух согласованных сторон. И что бы найти друг друга в многоквартирном доме одного адреса здания недостаточно, необходимо указать номер квартиры, в которой произойдет встреча. Так и на одной вычислительной машине может быть сразу несколько серверов, и клиенту, чтобы связаться с конкретным нужно указать ещё и номер порта по которому произойдет соединение. Итак, адрес и номер порта. **Адрес** подразумевает под собой идентификатор машины в пространстве сети Internet. Он может быть доменным именем, например, «javarush.ru», или обычным IP. **Порт** — уникальный номер, с которым связан определённый сокет (этот термин будет рассмотрен далее), проще говоря, его занимает определённая служба для того что бы по нему могли связаться с ней. Так что для того что бы произошла встреча как минимум двух объектов на территории одного (сервера) — хозяин местности (сервер) должен занять конкретную квартиру (порт) на ней (машине), а второй должен найти место встречи зная адрес дома (домен или ip), и номер квартиры (порт).

2 Знакомьтесь, Socket

Среди понятий и терминов, связанных с работой в сети, если одно очень важное – Сокет. Оно обозначает точку, через которую происходит соединение. Проще говоря, сокет соединяет в сети две программы. Класс **Socket** реализует идею сокета. Через его каналы ввода/вывода будут общаться клиент с сервером:



Объявляется этот класс на стороне клиента, а сервер воссоздаёт его, получая сигнал на подключение. Так происходит общение в сети. Для начала вот возможные конструкторы класса **Socket**:
Socket(String имя_хоста, int порт) throws UnknownHostException, IOException
Socket(InetAddress IP-адрес, int порт) throws UnknownHostException

«имя_хоста» — подразумевает под собой определённый узел сети, ip-адрес. Если класс сокета не смог преобразовать его в реальный, существующий, адрес, то сгенерируется исключение **UnknownHostException**. Порт — есть порт. Если в качестве номера порта будет указан 0, то система сама выделит свободный порт. Также при потере соединения может произойти исключение **IOException**. Следует отметить тип адреса во втором конструкторе — **InetAddress**. Он приходит на помощь, например, когда нужно указать в качестве адреса доменное имя. Так же когда под доменом подразумевается несколько ip-адресов, то с помощью **InetAddress** можно получить их массив. Тем не менее с ip он работает тоже. Так же можно получить имя хоста, массив байт составляющих ip адрес и т.д. Мы немного затронем его далее, но за полными сведениями придется пройти к официальной документации. При инициализации объекта типа **Socket**, клиент, которому тот принадлежит, объявляет в сети, что хочет соединиться с сервером про определённому адресу и номеру порта. Ниже представлены самые часто используемые методы класса **Socket**: InetAddress getInetAddress() – возвращает объект содержащий данные о сокете. В случае если сокет не подключен – null int getPort() – возвращает порт по которому происходит соединение с сервером int getLocalPort() – возвращает порт к которому привязан сокет. Дело в том, что «общаться» клиент и сервер могут по одному порту, а порты, к которым они привязаны – могут быть совершенно другие boolean isConnected() – возвращает true, если соединение установлено void connect(SocketAddress адрес) – указывает новое соединение boolean isClosed() – возвращает true, если сокет закрыт boolean isBound() - возвращает true, если сокет действительно привязан к адресу Класс **Socket** реализует интерфейс **AutoCloseable**, поэтому его можно использовать в конструкции **try-with-resources**. Тем не менее закрыть сокет также можно классическим образом, с помощью close().

3 ServerSocket

Допустим мы объявили, в виде класса **Socket**, на стороне клиента запрос на соединение. Как сервер разгадает наше желание? Для это сервер имеет такой класс как **ServerSocket**, и метод accept() в нём. Его конструкторы представлены ниже:
ServerSocket() throws IOException
ServerSocket(int порт) throws IOException
ServerSocket(int порт, int максимум_подключений) throws IOException
ServerSocket(int порт, int максимум_подключений, InetAddress локальный_адрес) throws IOException
При объявлении **ServerSocket** не нужно указывать адрес соединения, потому что общение происходит на машине сервера. Только при многоканальном хосте нужно указать к какому ip привязан сокет сервера.

3.1 Сервер, который говорит нет

Так как предоставлять программе больше ресурсов чем ей необходимо - и затратное и не разумное дело, поэтому в конструкторе **ServerSocket** вам предлагают объявить максимум соединений, принимаемых сервером при работе. Если оно не указано, то умолчанию это число будет считаться равным 50. Да, по идее можно предположить, что **ServerSocket** это такой же сокет, только для сервера. Но он играет совершенно иную роль нежели класс **Socket**. Он нужен только на этапе создания соединения. Создав объект типа **ServerSocket** необходимо

выяснить, что с сервером кто-то хочет соединиться. Тут подключается метод accept(). Искомый ждёт пока кто-либо не захочет подсоединится к нему, и когда это происходит возвращает объект типа **Socket**, то есть воссозданный клиентский сокет. И вот когда сокет клиента создан на стороне сервера, можно начинать двухстороннее общение. Создать объект типа **Socket** на стороне клиента и воссоздать его с помощью **ServerSocket** на стороне сервера – вот необходимый минимум для соединения.

4 Письмо "деду морозу"

Вопрос: Как конкретно общаются клиент и сервер? **Ответ:** Через потоки ввода вывода. Что мы уже имеем? Сокет с адресом сервера и номером порта у клиента, и тоже самое, благодаря accept(), на стороне сервера. Так что разумно предположить, что общаться они будут как раз через сокет. Для этого есть два метода которые дают доступ к потокам **InputStream** и **OutputStream** объекта типа **Socket**. Вот они:
InputStream getInputStream()
OutputStream getOutputStream()
Так как читать и писать голые байты не так эффективно - потоки можно обернуть в классы адаптеры, буферизированные, или нет. Например:
BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
BufferedWriter out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
Что бы общение было двунаправленным такие операции необходимо проделать на обеих сторонах. Теперь вы можете отослать что-то с помощью in, и принять с помощью out, и наоборот. Собственно, это практически единственная функция класса **Socket**. И да, не забывайте про метод flush() для **BufferedWriter** – он выталкивает содержимое буфера. Если этого не сделать, информация не будет передана, а, следовательно, не будет получена. Так же принимающий поток ждет указатель конца строки – «\n», иначе сообщение не будет принято, так как фактически сообщение не окончено, и не является целым. Если вам это кажется неудобным, не расстраивайтесь, всегда можно воспользоваться классом **PrintWriter**, которым нужно обернуть out, указать вторым аргументом true и тогда выталкивание из буфера будет происходить автоматически:
PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())), true);
Так же при этом указывать конец строки нет необходимости, за вас это делает данный класс. Но является ли ввод/вывод строк пределом возможностей сокета? Нет, хотите оправлять объекты через потоки сокета? Ради бога. Сериализуйте их, и вперед:
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());

5 Реальная связь по сети Internet

Так как для того что бы соединиться по реальной сети с реальным ip адресом нужно иметь полноценный сервер, а так как:

1. Наш будущий чат, как утилита, такими способностями не обладает. Он может лишь установить соединение и принять/отправить сообщение. То есть он не обладает реальными возможностями сервера.
2. Наш сервер, содержащий лишь данные сокета и потоков ввода/вывода, не может работать как реальный WEB- или FTP-сервер, то имея лишь это мы не сможем соединиться по сети Internet.

И к тому же, мы лишь начинаем разрабатывать программу, а это значит что она не достаточно стабильна чтобы сразу работать с реальной сетью, так что мы будем использовать как адрес для соединения локальный хост. То есть по идее клиент и сервер все равно никак не будут связаны кроме как через сокет, но для отладки программы они буду находиться на одной машине, без реального контакта по сети. Для того что бы указать в конструкторе **Socket**, что адрес локальный, существует 2 способа:

1. Написать в качестве аргумента адреса «localhost», означающий локальную заглушку. Так же для этого подходит «127.0.0.1» - это всего лишь цифровая форма заглушки.
2. С помощью InetAddress:
    1. InetAddress.getByName(null) - null указывает на локальный хост
    2. InetAddress.getByName("localhost")
    3. InetAddress.getByName("127.0.0.1")

Для простоты мы будем использовать «localhost» типа String. Но все остальные варианты тоже работоспособны.

6 Настало время для беседы

Итак, все что нам нужно для реализации сеанса разговора с сервером у нас уже есть. Осталось собрать это воедино: Следующий листинг показывает, как подключается клиент к серверу, отсылает ему одно сообщение, а тот в свою очередь подтверждает, что получил сообщение используя его как аргумент в своём: "Server.java"

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

public class Server {

    private static Socket clientSocket; //сокет для общения
    private static ServerSocket server; // серверсокет
    private static BufferedReader in; // поток чтения из сокета
```

```
private static BufferedWriter out; // поток записи в сокет

public static void main(String[] args) {
   try {
      try  {
         server = new ServerSocket(4004); // серверсокет прослушивает порт 4004
         System.out.println("Сервер запущен!"); // хорошо бы серверу
         //   объявить о своем запуске
         clientSocket = server.accept(); // accept() будет ждать пока
         //кто-нибудь не захочет подключиться
         try { // установив связь и воссоздав сокет для общения с клиентом можно перейти
            // к созданию потоков ввода/вывода.
            // теперь мы можем принимать сообщения
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            // и отправлять
            out = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));

            String word = in.readLine(); // ждём пока клиент что-нибудь нам напишет
            System.out.println(word);
            // не долго думая отвечает клиенту
            out.write("Привет, это Сервер! Подтверждаю, вы написали : " + word + "\n");
            out.flush(); // выталкиваем все из буфера

         } finally { // в любом случае сокет будет закрыт
            System.out.println("dfjkhgkdf");
            clientSocket.close();
            // потоки тоже хорошо бы закрыть
            in.close();
            out.close();
         }
      } finally {
         System.out.println("Сервер закрыт!");
            server.close();
```

```
            }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
"Client.java"
import java.io.*;
import java.net.Socket;

public class Client {

    private static Socket clientSocket; //сокет для общения
    private static BufferedReader reader; // нам нужен ридер читающий с консоли, иначе как
    // мы узнаем что хочет сказать клиент?
    private static BufferedReader in; // поток чтения из сокета
    private static BufferedWriter out; // поток записи в сокет

    public static void main(String[] args) {
        try {
            try {
                // адрес - локальный хост, порт - 4004, такой же как у сервера
                clientSocket = new Socket("localhost", 4004); // этой строкой мы запрашиваем
                //  у сервера доступ на соединение
                reader = new BufferedReader(new InputStreamReader(System.in));
                // читать соообщения с сервера
                in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                // писать туда же
                out = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));

                System.out.println("Вы что-то хотели сказать? Введите это здесь:");
                // если соединение произошло и потоки успешно созданы - мы можем
                //  работать дальше и предложить клиенту что то ввести
                // если нет - вылетит исключение
```

```
            String word = reader.readLine(); // ждём пока клиент что-нибудь
            // не напишет в консоль
            out.write(word + "\n"); // отправляем сообщение на сервер
            out.flush();
            String serverWord = in.readLine(); // ждём, что скажет сервер
            System.out.println(serverWord); // получив - выводим на экран
        } finally { // в любом случае необходимо закрыть сокет и потоки
            System.out.println("Клиент был закрыт...");
            clientSocket.close();
            in.close();
            out.close();
        }
    } catch (IOException e) {
        System.err.println(e);
    }

  }
}
```

Разумеется, запускать следует сначала сервер, ибо к чему будет подключаться клиент при запуске если не будет того что его подключит? :) Вывод будет такой: /* ***Вы что-то хотели сказать? Введите это здесь: Алло, сервер? Ты меня слышишь? Привет, это Сервер! Подтверждаю, вы написали : Алло, сервер? Ты меня слышишь? Клиент был закрыт...*** */ Ура! Мы научили сервер общаться с клиентом! Что бы общение происходило не в две реплики, а столько сколько угодно, просто оберните чтение и запись потоков в цикл while (true) и укажите для выхода что, по определённому сообщению, например, «exit», цикл прерывался, и программа завершилась бы.

7 Многопользовательский – лучше

То, что сервер нас слышит это хорошо, но куда лучше если можно было бы пообщаться с кем-то из себе подобных. Все исходники я приложу в конце статьи, так что здесь я буду показывать не всегда большие, но важные кусочки кода, которые дадут возможность при правильном использовании состряпать многопользовательский чат. Итак, мы хотим, чтобы через сервер мы могли общаться с каким-то другим клиентом. Как это сделать? Очевидно, что раз клиентская программа имеет свой метод main, то значит его можно запускать отдельно от сервера и параллельно с другими клиентами. Что нам это дает? Каким-то образом нужно что бы при каждом новом подключении сервер не переходил сразу к общению, а записывал это соединение в какой-то список и переходил к ожиданию нового подключения, а общением с конкретным клиентом занимался бы какой-то вспомогательный сервис. Да и клиенты должны писать на сервер и ждать ответа независимо

друг от друга. На помощь приходят нити. Допустим у нас есть класс, отвечающий за запоминание новых подключений: У него должны быть указаны:

1. Номер порта.
2. Список, в который он записывает новое соединение.
3. И **ServerSocket**, в единственном (!) экземпляре.

```
public class Server {

   public static final int PORT = 8080;
   public static LinkedList<ServerSomthing> serverList = new LinkedList<>(); // список всех нитей

   public static void main(String[] args) throws IOException {
      ServerSocket server = new ServerSocket(PORT);
         try {
         while (true) {
            // Блокируется до возникновения нового соединения:
            Socket socket = server.accept();
            try {
               serverList.add(new ServerSomthing(socket)); // добавить новое соединенние в список
            } catch (IOException e) {
               // Если завершится неудачей, закрывается сокет,
               // в противном случае, нить закроет его при завершении работы:
               socket.close();
            }
         }
      } finally {
         server.close();
      }
   }
}
```

Окей, теперь каждый воссозданный сокет не потеряется, а будет храниться на сервере. Дальше. Каждого клиента должен кто-то слушать. Давайте создадим нить с серверными функциями из прошлой главы.

```
class ServerSomthing extends Thread {

    private Socket socket; // сокет, через который сервер общается с клиентом,
    // кроме него - клиент и сервер никак не связаны
    private BufferedReader in; // поток чтения из сокета
    private BufferedWriter out; // поток записи в сокет

    public ServerSomthing(Socket socket) throws IOException {
        this.socket = socket;
        // если потоку ввода/вывода приведут к генерированию исключения, оно проброситься дальше
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        out = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        start(); // вызываем run()
    }
    @Override
    public void run() {
        String word;
        try {

            while (true) {
                word = in.readLine();
                if(word.equals("stop")) {
                    break;              }
                for (ServerSomthing vr : Server.serverList) {
                    vr.send(word); // отослать принятое сообщение с
                  // привязанного клиента всем остальным включая его
                }
            }

        } catch (IOException e) {
        }
    }
```

```
    private void send(String msg) {
        try {
            out.write(msg + "\n");
            out.flush();
        } catch (IOException ignored) {}
    }
}
```

Итак, в конструкторе серверной нити должен быть инициализирован сокет, через который нить будет общаться с конкретным клиентом. Также потоки ввода/вывода, и ко всему прочему нужно запустить нить прямо из конструктора. Хорошо, но что будет происходить при чтении сообщения от клиента для серверной нити? Отсылать обратно только своему клиенту? Не очень-то эффективно. Мы делаем многопользовательский чат, поэтому нам нужно что бы каждый подключенный клиент получил то что написал кто-то один. Нужно воспользоваться списком всех серверных нитей, привязанных к своим клиентам, и отослать каждое присланное конкретной нити сообщение, что бы та отослала его своему клиенту:

```
for (ServerSomthing vr : Server.serverList) {
    vr.send(word); // отослать принятое сообщение
    // с привязанного клиента всем остальным, включая его
}
private void send(String msg) {
    try {
        out.write(msg + "\n");
        out.flush();
    } catch (IOException ignored) {}
}
```

Теперь все клиенты узнают то, что сказал один из них! Если вы не хотите, чтобы сообщение приходило тому, кто его отправил (он и так знает, что он написал!) просто при переборе нитей укажите что бы при обработке объекта **this** цикл переходил к следующему элементу, не выполняя над ним никаких действий. Или же, если хотите, отправьте сообщение клиенту, в котором написано, что сообщение успешно принято и разослано. С сервером теперь все понятно. Перейдём к клиенту, а точнее к клиентам! Там все так же, по аналогии с клиентом из прошлой главы, только создавая экземпляр нужно как было показано в данной главе с сервером, создать все необходимое в конструкторе. Но что если при создании клиента он ещё не успел ничего ввести, а ему уже что-то отправили? (Например, историю переписки тех, кто уже подключился к чату до него). Так что циклы, в которых буду обрабатываться присланные сообщения должны быть отделены от тех в которых читаются сообщения с консоли и отправляются на сервер для пересылки остальным. На помощь снова приходят нити. Нет смысла создавать клиента как нить. Удобнее сделать нить с циклом в методе run читающую сообщения, а также по аналогии - пишущую:

```
// нить чтения сообщений с сервера
```

```
private class ReadMsg extends Thread {
    @Override
    public void run() {

        String str;
        try {
            while (true) {
                str = in.readLine(); // ждем сообщения с сервера
                if (str.equals("stop")) {

                    break; // выходим из цикла если пришло "stop"
                }
                        }
        } catch (IOException e) {

        }
    }
}
// нить отправляющая сообщения приходящие с консоли на сервер
public class WriteMsg extends Thread {

    @Override
    public void run() {
        while (true) {
            String userWord;
            try {
                userWord = inputUser.readLine(); // сообщения с консоли
                if (userWord.equals("stop")) {
                    out.write("stop" + "\n");
                    break; // выходим из цикла если пришло "stop"
                } else {
                    out.write(userWord + "\n"); // отправляем на сервер
                }
```

```
                out.flush(); // чистим
            } catch (IOException e) {

            }

        }
    }
}
```

В конструкторе клиента необходимо просто запустить эти нити. А как правильно закрыть ресурсы клиента если тот захочет выйти? Нужно ли закрывать ресурсы серверной нити? Для этого необходимо будет скорее всего создать отдельный метод, вызывающийся при выходе из цикла обработки сообщений. Там нужно будет закрыть сокет и потоки ввода/вывода. Тот же сигнал окончания сессии для конкретного клиента должен быть отправлен его серверной нити, которая должна сделать тоже со своим сокетом и удалить себя из списка нитей в основном классе сервера.

8 Нет предела совершенству

Можно бесконечно долго выдумывать новые фичи для совершенствования своего проекта. Но что точно должно быть передано ново подключившемуся клиенту? Я думаю, что последние десять событий, произошедших до его прихода. Для это необходимо создать класс, в котором в объявленный список будет заноситься последнее действие с любой серверной нитью, и, если список уже полон (то есть 10 уже есть), удалить первое и занести последним пришедшее. Для того что бы содержимое этого списка получил новый подключившийся, нужно при создании серверной нити, в потоке вывода, отослать их клиенту. Как это сделать? Например, так:

```
public void printStory(BufferedWriter writer) {
// ...
}
```

Серверная нить уже создала потоки и может поток вывода передать как аргумент. Далее просто нужно в цикле перебора все что необходимо передать новому клиенту.

Заключение:

Это лишь основы основ, и скорее всего такая архитектура чата не подойдёт при создании реального приложения. Эта программа создана в учебных целях и на её основе я показал, как можно заставить общаться клиента с сервером (и наоборот), как это сделать для нескольких подключений, и, конечно же, как это организовано на сокетах. Ниже переставлены источники, а так же приложен исходный код разбираемой программы.