Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

KATEDRA INŻYNIERII OPROGRAMOWANIA http://kio.wi.zut.edu.pl/

INŻYNIERIA OPROGRAMOWANIA

Zaawansowane Techniki Programowania Java

#03 : NETWORKING (*java.net*)

Prowadzący:

Krzysztof Kraska

email: kkraska@wi.zut.edu.pl

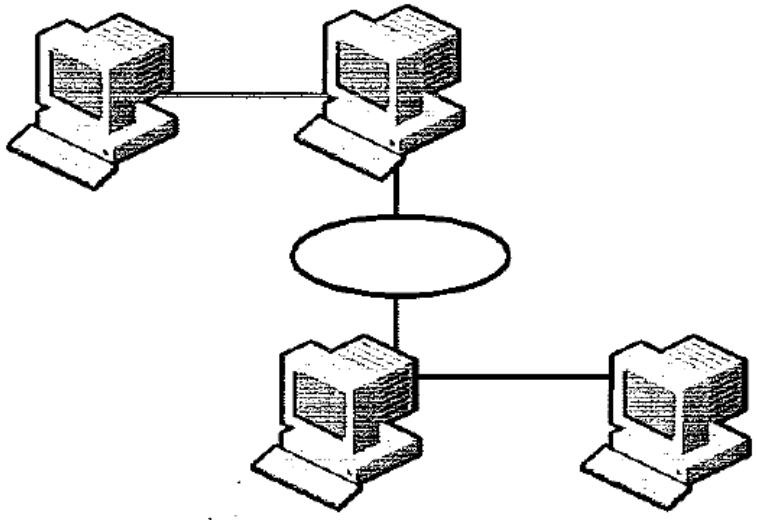
Szczecin, 5 kwietnia 2018 r.

The java.net Package

- **InetAddress** -- Represents Internet address. Provides functions to query hostnames, and so forth.
- **Socket** (a client socket) -- Communication endpoint
- **ServerSocket** -- Listens for client requests
- **DatagramSocket** -- Uses individually addressed and routed ***DatagramPacket*** instances
- **MulticastSocket** -- For IP multicast packets
- **DatagramPacket** -- Connectionless delivery object
- **URL** -- Represents a pointer to a network resource

Networking with Java

- Program at the application layer
- Application requirements may dictate which communications protocol to use
- At run time applications communicate over ports



Transmission Control Protocol (TCP)

Point-to-point connection-based channel for applications that require reliable communications, TCP guarantees delivery. Order of data that is transported and received is important to the application's success.

Examples –

FTP (File Transfer Protocol)

HTTP (HyperText Transfer Protocol)

SMTP (Simple Mail Transfer Protocol)

Telnet

User Datagram Protocol (UDP)

Connectionless-based communication that is not guaranteed between applications on the network. UDP sends independent packets of data, called datagrams, from one application to another in no specific order.

Examples –

DNS (Domain Name Service)

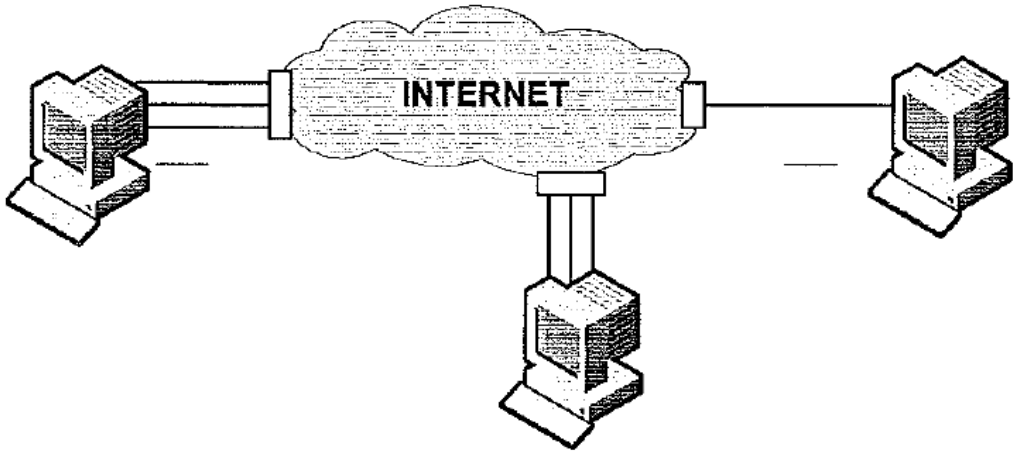
RIP (Routing Information Protocol)

SNMP (Simple Network Management Protocol)

Ports

Computers can have multiple network applications running simultaneously, thus the application's port must be used to identify which application to access.

Port – represents the address of a specific application



Application Endpoint Naming

- To communicate we need to uniquely identify a host and a port. (This identifies a destination)
- An InetAddress object can be used to specify a host address and comes in two forms:
  - Dotted ASCII, for example www.ibm.com
  - Dotted Decimal, for example 127.0.0.1
- The InetAddress class has no public constructors. It uses factory methods instead.

```
InetAddress inetAddr = InetAddress.getByName(String);
InetAddress inetAddr = InetAddress.getLocalHost();
InetAddress[] inetAddr = InetAddress.getAllByName(String);
```

Application Endpoint Naming

- Get Internet address:

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

returns an `InetAddress` object that encapsulates the sequence of four bytes `132.163.4.104`.

- Access the bytes:

```
byte[] addressBytes = address.getAddress();
```

- Get all hosts with the host name:

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

- Get address of the local host:

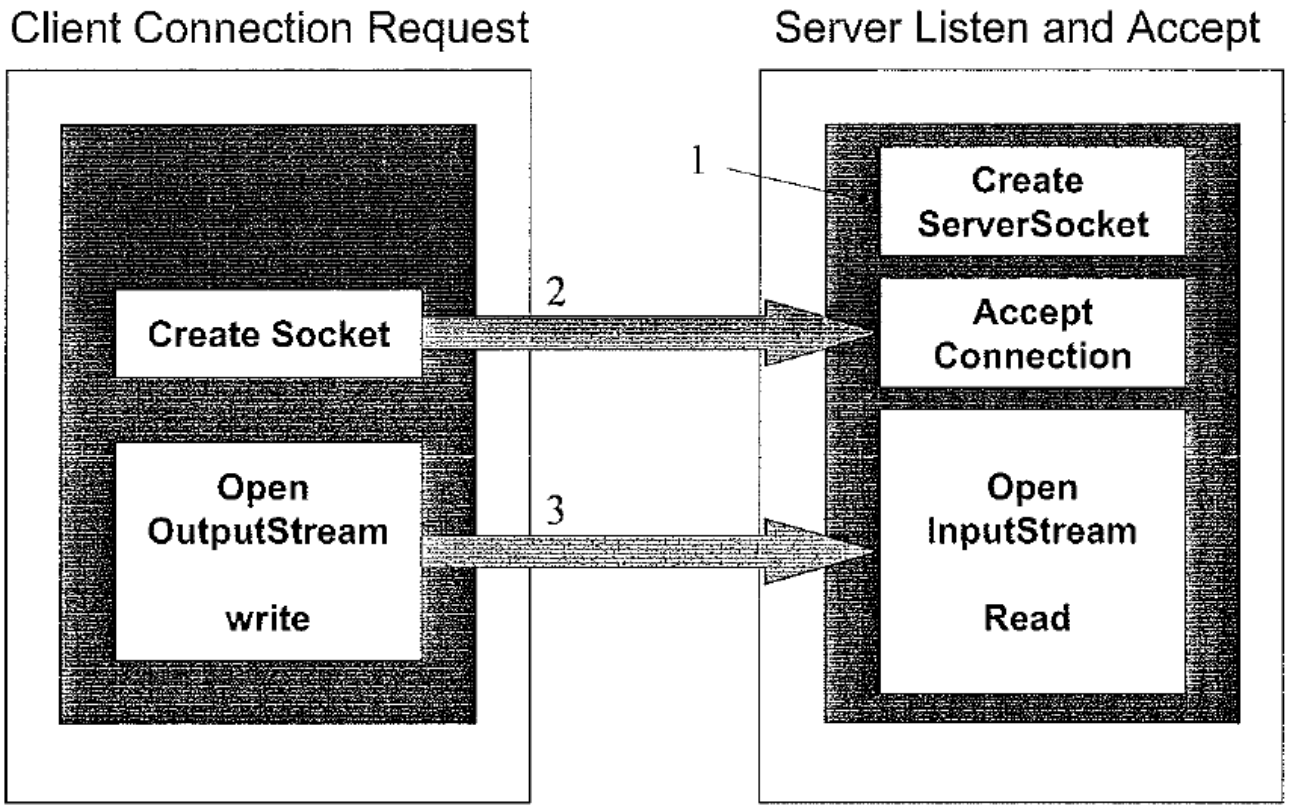
```
InetAddress address = InetAddress.getLocalHost();
```

NETWORKING

Sockets

- A socket is one of the end-points in a two-way communication link between two programs running on the network.
- Java implements Socket classes to represent a client / server connectivity environment.
- Two broad categories of Sockets based on the nature of communication:
  - Stream Socket is used to maintain a Connection between the two endpoints, offers reliability guarantees and is bi-directional (TCP)
  - Datagram Socket is used for one-shot one-way messages, and offers no reliability guarantees (UDP)

Connecting With Sockets



The server is reading from an InputStream

NETWORKING

In a socket environment, the terms client and server are used loosely. A computer can be a client and a server at the same time. The term server is used to recognize the computer that is being connected to. Client refers to the computer that is initiating the conversation.

Socket connection process:

1. A ServerSocket must be created, it listens to a specific port
2. A client then creates a Socket that connects to a particular host and port
3. The ServerSocket accepts the connection and creates a Socket. We now have out two endpoints for communication
4. Depending on what the computers want to do, OutputStreams and InputStreams are opened that are used to pass data.

Server-side Socket

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

public class ServerSideSocket {

  public static void main (String[] args) throws Exception{
         // press Ctrl-C to terminate this program
        ServerSocket ss = new ServerSocket(1111); // specify port number.
        System.out.println("Server listening at " +
                           InetAddress.getLocalHost() +
                           " on port " + ss.getLocalPort());
        while (true) {
            Socket          s       = ss.accept();//wait for new client to call
            DataInputStream dis     = new DataInputStream(s.getInputStream());
            String          message = dis.readUTF();// wait for client to send
            System.out.println(message);
             dis.close();
        }

  }
}
```

NETWORKING

Client-side Socket

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

public class ClientSideSocket {

  public static void main (String[] args) throws Exception {
    if (args.length != 3) {
        System.out.println("Usage: java ClientSide Hostname port
  message");
        System.exit(0);
    }

     String serverName = args[0]; // either ASCII or numeric form is OK
     int    serverPort = Integer.parseInt(args[1]);
     String message    = args[2];


     Socket s = new Socket(serverName, serverPort);//wait for server to
   accept
     DataOutputStream dos = new DataOutputStream(s.getOutputStream());
     dos.writeUTF(message);
     dos.close(); // and flush

  }
}
```

Datagram Sockets

- A Datagram Socket is the sending or receiving point for a packet delivery service (Datagrams) using UDP.
- Each packet sent or received on a Datagram Socket is individually addressed and routed.
- Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

Server-side Datagram Socket

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

public class ServerDGSocket {
    // press Ctrl-C to terminate this program
    public static void main (String[] args) throws Exception {
          byte[] tempBuffer = new byte[256];
          DatagramSocket ds = new DatagramSocket(1111); // specify port
    number.
           System.out.println("Server listening at " +
    InetAddress.getLocalHost()
                                  " on port " + ds.getLocalPort());
          while (true) { // receive request
            DatagramPacket packet = new DatagramPacket(tempBuffer,
                                                  tempBuffer.length);
             ds.receive(packet);
             String receiveMessage = new String(packet.getData());
             receiveMessage = receiveMessage.substring(0,
    packet.getLength());

             System.out.println("Received " + receiveMessage + " from port
    "
                           + packet.getPort() + " at " +
    packet.getAddress());
           }
         ds.close();
      }
}
```

Client-side Datagram Socket

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

public class ClientSideDGSocket {
    public static void main (String[] args) throws Exception {
          if (args.length != 3){
              System.out.println("Usage: java ClientSide Hostname port
    message");
              System.exit(0);
          }

          String serverName = args[0]; // either ASCII or numeric form is OK
          int    serverPort = Integer.parseInt(args[1]);
          String message    = args[2];

          DatagramSocket socket = new DatagramSocket(); // get a datagram
    socket
          byte[] buf = message.getBytes(); // send request
          InetAddress address = InetAddress.getByName(args[0]);
          DatagramPacket packet = new DatagramPacket(buf, buf.length,
    address,
    serverPort);
          socket.send(packet);
          socket.close();

    }
}
```

Serving Multiple Clients

Typically, a server runs constantly on a server computer, and multiple clients from all over the Internet might want to use it at the same time.

Every time the program has established a new socket connection (the call to `accept()` returns a socket) it will launch a new thread to take care of the connection.

```
public class ThreadedEchoServer {
     public static void main(String… args) {
          try {
               int i = 1;
               ServerSocket s = new ServerSocket(8189);
               while (true) {
                    Socket incoming = s.accept();
                    System.out.println("Spawning " + i);
                    Runnable r = new ThreadedEchoHandler(incoming);
                    Thread t = new Thread(r);
                    t.start( );
                    i++;
               }
          }
          catch (IOException e) {
               e.printStackTrace();
          }
     }
}
```

Serving Multiple Clients

```
class ThreadedEchoHandler implements Runnable {
      private Socket incoming;

      public ThreadedEchoHandler(Socket i) {
            incoming = i;
      }

      public void run() {
            try {
                  try {
                        InputStream inStream = incoming.getInputStream();
                        OutputStream outStream = incoming.getOutputStream();

                        Scanner in = new Scanner(inStream);
                        PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
                        out.println( "Hello! Enter BYE to exit." );

                        boolean done = false;
                        while (!done && in.hasNextLine()) {
                              String line = in.nextLine();
                              out.println("Echo: " + line);
                              if (line.trim().equals("BYE")) done = true;
                        }
                  }
                  finally {
                        incoming.close();
                  }
            }
            catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Half-Close

The *half-close* provides the ability for one end of a `java.net.Socket` connection to terminate its output while still receiving data from the other end.

- sets the output stream to “end of stream.”
  `void shutdownOutput()`
- sets the input stream to “end of stream.”
  `void shutdownInput()`
- returns true if output has been shut down.
  `boolean isOutputShutdown()`
- returns true if input has been shut down
  `boolean isInputShutdown()`

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// send request data
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// now socket is half-closed
// read response data
while (in.hasNextLine() != null) { String line = in.nextLine(); . . . }
socket.close();
```

Multicast Sockets

- Used to send Datagram packets to multiple clients that have joined a particular Multicast group
- Group of hosts sharing a common class D multicast address, in the range 224.0.0.0 to 239.255.255.255
- Network routers forward packets to all in the Multicast group
- Can be a large security hole since packets are forwarded many times

Server-side Multicast Socket

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

public class ServerMulticastSocket{
    // press Ctrl-C to terminate this program
    public static void main (String[] args)throws Exception {

         MulticastSocket socket = new MulticastSocket();
         InetAddress group = InetAddress.getByName("226.0.0.1");
           while (true) {
              byte[] buffer = new byte[1024];
               // don't wait for request...just send a quote
               String dString = new Date().toString();
               buffer = dString.getBytes();
               DatagramPacket packet = new DatagramPacket(buffer,
    dString.length(),
                              group, 4444);
               System.out.println("Sending the packages....");
               socket.send(packet,(byte)1);
               java.lang.Thread.sleep((long)3000);
           }
          socket.close();
    }
}
```

Client-side Multicast Socket

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

public class ClientMulticastSocket {
   public static void main (String[] args)throws IOException {
       MulticastSocket socket = new MulticastSocket(4444);
       InetAddress address = InetAddress.getByName("226.0.0.1");
       socket.joinGroup(address);
       // get a few quotes
       for (int i = 0; i < 10; i++) {

             byte[] buf = new byte[1024];
            DatagramPacket packet = new DatagramPacket(buf, buf.length);
            System.out.println("Waiting for package to arrive....");
            socket.receive(packet);

            String received = new String(packet.getData());
            System.out.println("Quote of the Moment: " + received);
        }
        socket.leaveGroup(address);
        socket.close();
   }
}
```

URL Naming for Network Resources

- URL: **U**niform **R**esource **L**ocator
- Names a network resource (For example, Web, DBMS, directory, file, application, …)
- May encapsulate protocol, host, port, path, authentication, type, parameters
- Represented by class java.net.URL

NETWORKING

URLS are made up with the following parts:

www.http://HostName:Port/FileName/Reference

Protocol - The protocol.

Host Name - The name (or address) of the machine on which the resource lives.

Port - The port number to which to connect (typically optional).

File Name - The pathname to the file on the machine.

Reference - A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

Creating a URL Object

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    . . .
    // exception handler code here
    . . .
}
```

Java.net.URL constructors

URL(String spec)
Creates a URL object from the String representation.

URL(String protocol, String host, int port, String file)
Creates a URL object from the specified protocol, host, port number, and file.

URL(String protocol, String host, int port, String file, URLStreamHandler handler)
Creates a URL object from the specified protocol, host, port number, file, and handler.

URL(String protocol, String host, String file)
Creates a URL from the specified protocol name, host name, and file name.

URL(URL context, String spec)
Creates a URL by parsing the given spec within a specified context.

URL(URL context, String spec, URLStreamHandler handler)
Creates a URL by parsing the given spec with the specified handler within a specified context.

Parsing a URL Object

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

public class URLParse {

  public static void main(String[] args) throws Exception {
     URL aURL = new

   URL("http://ibm.com:80/services/learning/training.html");
     System.out.println("protocol = " + aURL.getProtocol());
     System.out.println("host = " + aURL.getHost());
     System.out.println("port = " + aURL.getPort());
     System.out.println("filename = " + aURL.getFile());
     System.out.println("ref = " + aURL.getRef());
    }
}
```

Reading Contents of a URL Target

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

public class URLRead {
   public static void main(String[] args) throws Exception {
     URL site = new URL("http://www.ibm.com/");
     BufferedReader inBuffer = new BufferedReader(new
        InputStreamReader(site.openStream()));

     String line;
     while ((line = inBuffer.readLine()) != null)
        System.out.println(line);
     inBuffer.close();
    }
}
```

Connecting to a URL

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

public class URLConnect {
    public static void main (String args[]) throws Exception
    {
          URL url = new URL("http://www.ibm.com/Search?v=11&lang=en&cc=us&q=" +
                            args[0] + "&Search.x=19&Search.y=10");
           URLConnection connection = url.openConnection();
           connection.setDoInput(true);
           InputStream in = connection.getInputStream();
           // read reply
           StringBuffer b = new StringBuffer();
           BufferedReader r = new BufferedReader(new InputStreamReader(in));
           String line;
           while ((line = r.readLine()) != null) {
              b.append(line);
          }
           String s = b.toString();

           BufferedWriter writer = new BufferedWriter(new
   FileWriter("output.html"));
          writer.write(s);
          r.close();
    }
```

Using a URLConnection to Retrieve Information

- Call the `openConnection()` of the `URL` class to obtain the `URLConnection` object

```
URLConnection connection = url.openConnection();
```

- Set any request properties

```
setDoInput()
setDoOutput()
setIfModifiedSince()
setUseCaches()
setAllowUserInteraction()
setRequestProperty()
setConnectTimeout()
setReadTimeout()
```

- Connect to the remote resource by calling the connect method

```
connection.connect();
```

Using a URLConnection to Retrieve Information

- After connecting to the server, you can query the header information

```
getHeaderFieldKey() and getHeaderField()
getHeaderFields()

getContentType()
getContentLength()
getContentEncoding()
getDate()
getExpiration()
getLastModified()
```

- Finally, you can access the resource data

```
getInputStream()
getOutputStream()
getErrorStream()
```

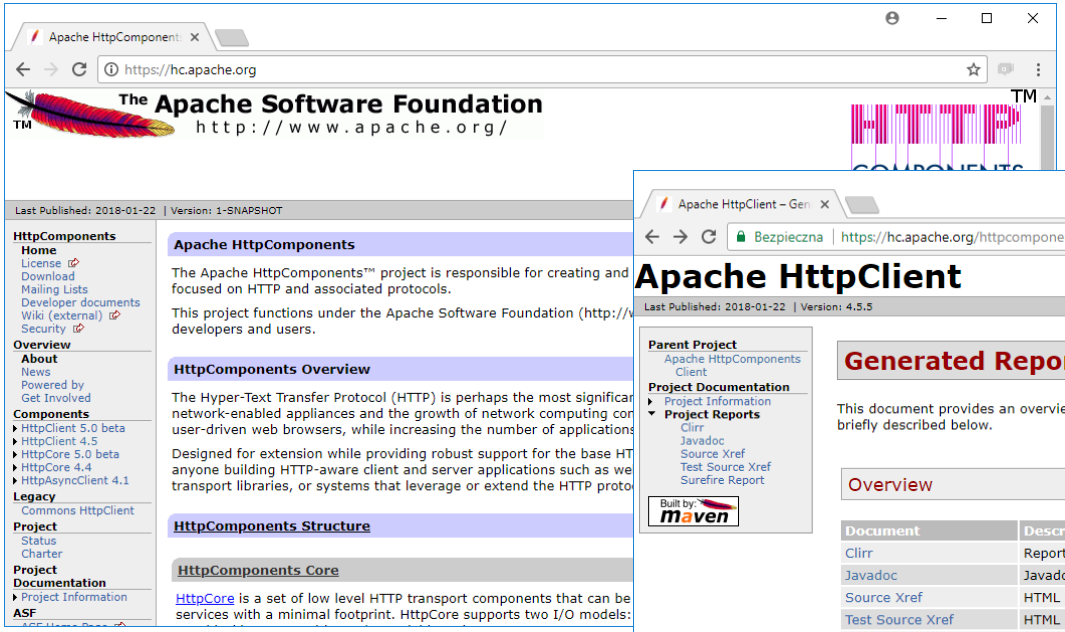
By default, the connection yields an input stream for reading but no output stream for writing. If you want an output stream (for example, for posting data to a web server), you need to call:

```
connection.setDoOutput(true);
```

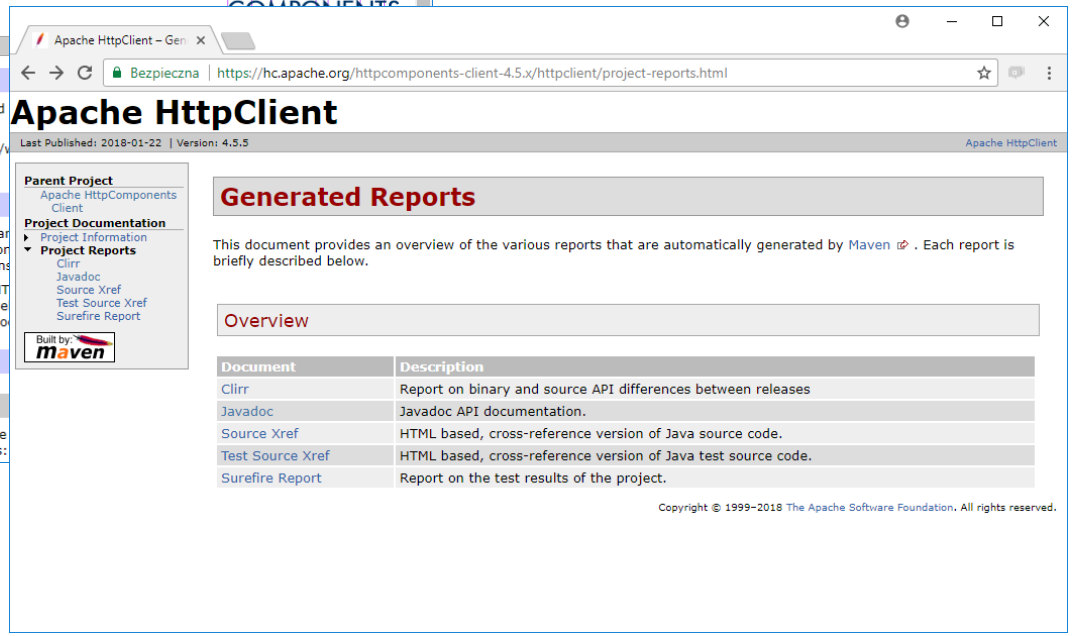
HttpURLConnection

```
String message = URLEncoder.encode("my message", "UTF-8");

try {
    URL url = new URL("http://www.example.com/resource");

    // A new connection is opened every time by calliing the openConnection method.
    HttpURLConnection connection = (HttpURLConnection)url.openConnection( );

    connection.setDoOutput(true);

    connection.setRequestMethod("POST");

    OutputStreamWriter writer = new OutputStreamWriter(connection.getOutputStream());

    writer.write("message=" + message);

    // Closes this output stream and releases any system resources
    writer.close();

    if (connection.getResponseCode() == HttpURLConnection.HTTP_OK) {
         // ...
    } else {
        // Server returned HTTP error code.
    }
} catch (MalformedURLException e) {
    // ...
} catch (IOException e) {
    // ...
}
```

Apache HTTP Components



- **Apache HTTP Core**



- **Apache HTTP Client**

- **Java Apache HttpClient REST (RESTful) client examples**

  https://alvinalexander.com/java/java-apache-httpclient-restful-client-examples

KONIEC

DZIĘKUJĘ ZA UWAGĘ!!!