

Middleware and Web Services

Lecture 2: Application Protocols

doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/mdw>



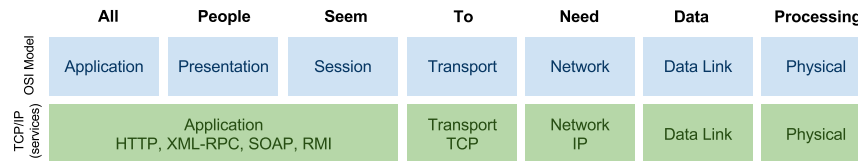
Modified: Tue Oct 03 2017, 19:43:13
Humla v0.3

Overview

- **Introduction to Application Protocols**
 - *Synchronous and Asynchronous Communication*
 - *Selected Networking Concepts*
- Introduction to HTTP

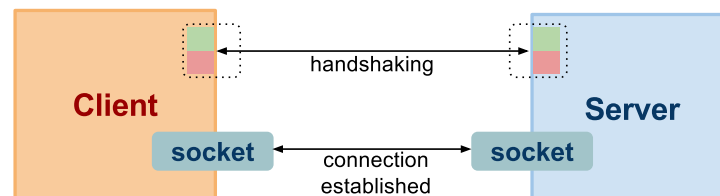
Application Protocols

- Remember this



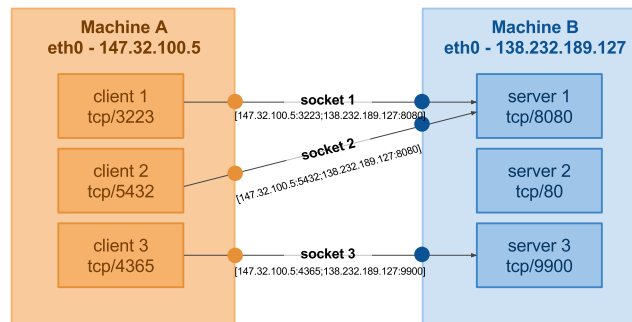
- App protocols mostly on top of the TCP Layer
 - use TCP socket for communication
- Major protocols
 - HTTP – most of the app protocols layered on HTTP
 - wide spread, but: implementors often break HTTP semantics
 - RMI – Remote Method Invocation
 - Java-specific, rather interface
 - may use HTTP underneath (among other things)
 - XML-RPC – Remote Procedure Call and SOAP
 - Again, HTTP underneath
 - WebSocket – new protocol part of HTML5

Socket



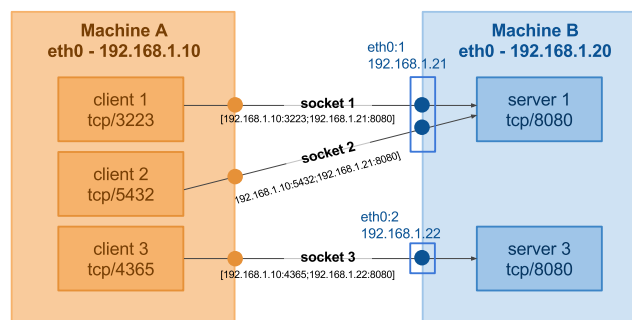
- Handshaking (connection establishment)
 - The server listens at `[dst_ip, dsp_port]`
 - Three-way handshake:
 - the client at `[src_ip, src_port]` sends a connection request
 - the server responds
 - the client acknowledges the response, can send data along
 - Result is a socket (virtual communication channel) with unique identification:
 `socket=[src_ip,src_port;dst_ip,dst_port]`
- Data transfer (resource usage)
 - Client/server writes/reads data to/from the socket
 - TCP features: reliable delivery, correct order of packets, flow control
- Connection close

Addressing in Application Protocol



- IP addressing: IP is an address of a machine interface
 - A machine can have multiple interfaces (`eth0`, `eth1`, `bond0`, ...)
- TCP addressing: TCP port is an address of an app running on a machine and listening on a machine interface
 - Multiple applications with different TCP ports may listen on a machine interface
- Application addressing
 - Additional mechanisms to address entities within an application
 - They are out of scope of IP/TCP, they are app specific
 - for example, Web apps served by a single Web server

Virtual IP



- Virtual IP
 - Additional IP addresses assigned to a network interface
 - For example, `eth0` – `eth0:1`, `eth0:2`, `eth0:3`, ...
 - A process can bind to the virtual IP
 - Multiple processes can listen on the same tcp port but on different virtual IPs
- Benefits
 - Floating IP – a process can move transparently to another physical machine
 - Network configuration can be preserved, no need to reconfigure
 - Failover concept uses floating IPs

Virtual IP Configuration

- Steps to configure virtual IP in Linux (example for **eth0**)

1. Find out the interface's network mask

```
1 | $ ifconfig eth0
2 | eth0      Link encap:Ethernet  HWaddr 00:0C:29:AB:5E:6A
3 | inet addr:172.16.169.184  Bcast:172.16.169.255  Mask:255.255.255.0
4 | ...
```

2. Create virtual IP using **ifconfig**

- it should use the same network mask

- it should be free, usually allocated to be used as a virtual IP

```
5 | $ sudo ifconfig eth0:1 172.16.169.184 netmask 255.255.255.0
6 | $ ifconfig eth0:1
7 | eth0:1    Link encap:Ethernet  HWaddr 00:0C:29:AB:5E:6A
8 | inet addr:172.16.169.186  Bcast:172.16.169.255  Mask:255.255.255.0
```

3. Update neighbours' ARP (Address Resolution Protocol) caches

- to associate the virtual IP with MAC address of **eth0**

- when the virtual IP was in use on other node or interface

```
9 | $ sudo arping -q -U -c 3 -I eth0 172.16.169.184
```



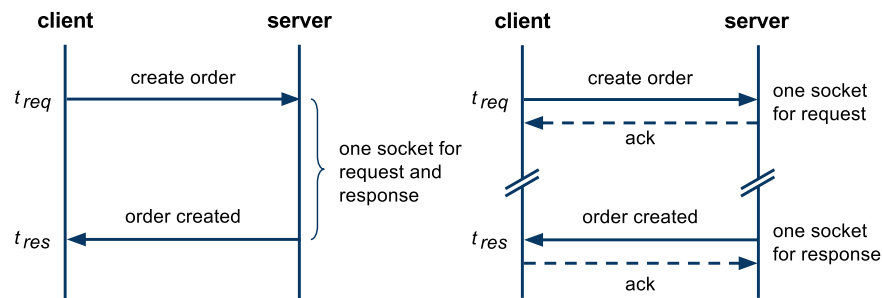
Tasks

- Configure a virtual IP on your computer and test it using **ping**

Overview

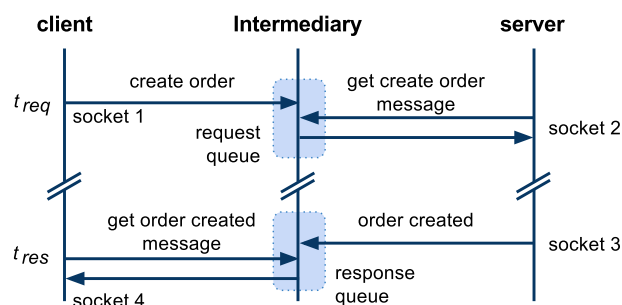
- Introduction to Application Protocols
 - *Synchronous and Asynchronous Communication*
 - *Selected Networking Concepts*
- Introduction to HTTP

Synchronous and Asynchronous Communication



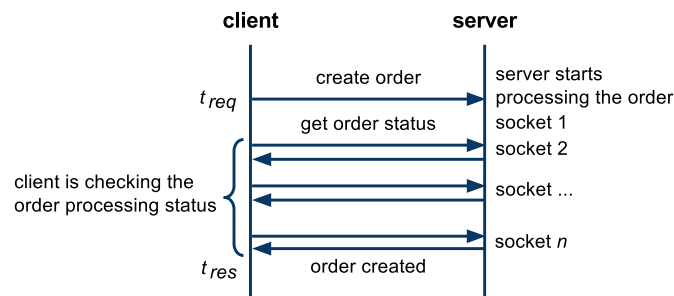
- **Synchronous**
 - one socket, $|t_{req} - t_{res}|$ is small
 - easy to implement and deploy, only standard firewall config
 - only the server defines endpoint
- **Asynchronous**
 - request, response each has socket, client and server define endpoints
 - $|t_{req} - t_{res}|$ can be large (hours, even days)
 - harder to do across network elements (private/public networks issue)

Asynchronous via Intermediary



- **Intermediary**
 - A component that decouples a client-server communication
 - It increases reliability and performance
 - The server may not be available when a client sends a request
 - There can be multiple servers that can handle the request
- **Further Concepts**
 - Message Queues (MQ) – queue-based communication
 - Publish/Subscribe (P/S) – event-driven communication

Asynchronous via Polling

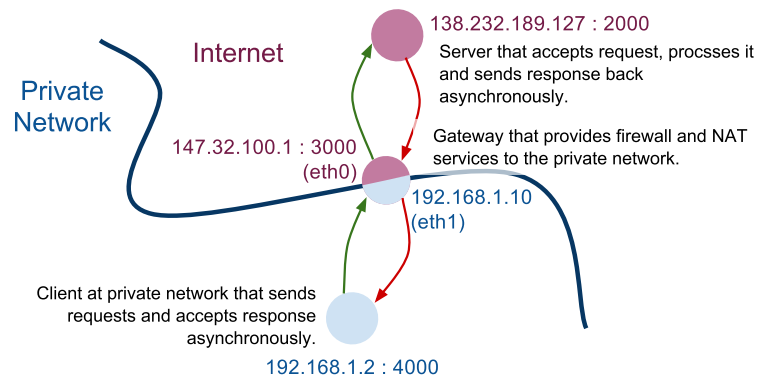


- Polling – only clients open sockets
 - A client performs multiple request-response interactions
 - The first interaction initiates a process on the server
 - Subsequent interactions check for the processing status
 - The last interaction retrieves the processing result
- Properties of environments
 - A server cannot open a socket with the client (network restrictions)
 - Typically on the Web (a client runs in a browser)

Overview

- Introduction to Application Protocols
 - Synchronous and Asynchronous Communication
 - *Selected Networking Concepts*
- Introduction to HTTP

Public/Private Network Configuration



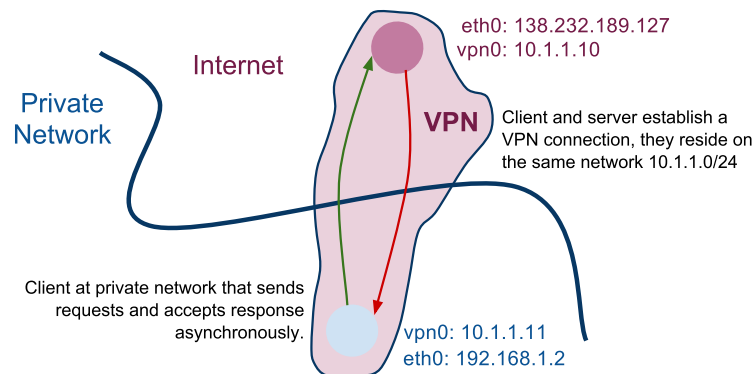
- Adds complexity to configuration of application
 - Config example at server with **eth0 = 147.32.100.1** (iptables)

```

1 | # enable ip forwarding from one interface to another within linux core
2 | echo 1 > /proc/sys/net/ipv4/ip_forward
3 |
4 | # redirect all communication coming to tcp/3000 to 192.168.1.2:4000
5 | iptables -t nat -A PREROUTING -i eth0 -p tcp --dport 3000 -j DNAT \
6 |   --to-dest 192.168.1.2 --to-port 4000

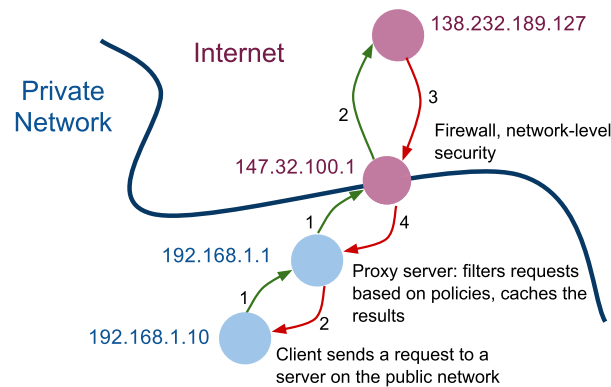
```

Virtual Private Network



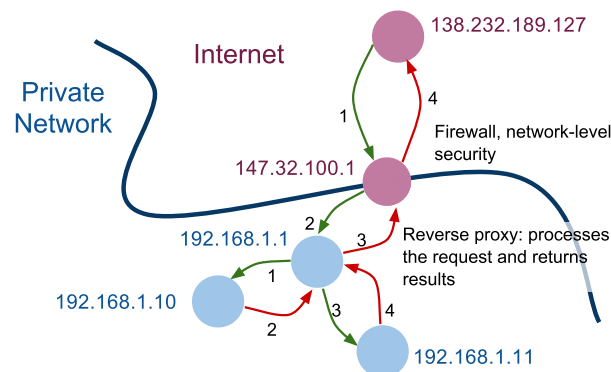
- VPN = Virtual Private Network
 - an overlay network between a client and a server
 - the network spans accross underlying network elements
 - Example:
 - VPN client starts a VPN connection with the VPN server via network interfaces
 - VPN server assigns an IP address to the VPN client from the server's subnet
 - Packets in VPN communication are encrypted and sent out in an outer VPN

Proxy Server



- Proxy Server
 - Centralized access control based on content
 - Performs request on behalf of the client
 - Caches content to increase performance, limits network traffic
 - Filters requests based on their destinations
 - Widely used in private networks in companies
 - Most of the proxy servers today are Web proxy servers

Reverse Proxy Server



- Reverse Proxy Server
 - Aggregates multiple request-response interactions with back-end systems
 - Processes the request on behalf of the client
 - Provides additional values to communication
 - Data transformations
 - Security – authentication, authorization
 - Orchestration of communication with back-end systems
 - Examples: Enterprise Service Bus, Security Gateway

Overview

- Introduction to Application Protocols
- **Introduction to HTTP**
 - *State Management*

Hypertext Transfer Protocol – HTTP

- Application protocol, basis of Web architecture
 - *Part of HTTP, URI, and HTML family*
 - *Request-response protocol*
- One socket for single request-response
 - *original specification*
 - *have changed due to performance issues*
 - *many concurrent requests*
 - *overhead when establishing same connections*
 - *HTTP 1.1 offers persistent connection and pipelining*
- HTTP is stateless
 - *Multiple HTTP requests cannot be normally related at the server*
 - *"problems" with state management*
 - *REST goes back to the original HTTP idea*

HTTP Request and Response

- Request Syntax

```
method uri http-version <crLf>
(header : value <crLf>)*
<crLf>
[ data ]
```

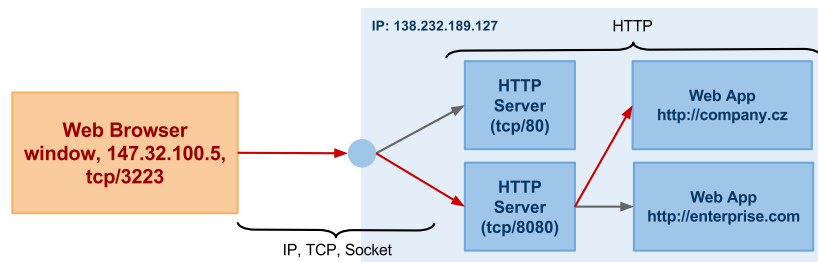
- Response Syntax

```
http-version response-code [ message ] <crLf>
(header : value <crLf>)*
<crLf>
[ data ]
```

- Semantics of terms

```
method          = "GET" | "POST" | "DELETE" | "PUT" | "HEAD" | "OPTIONS"
uri             = [ path ] [ ";" params ] [ "?" query ]
http-version    = "HTTP/1.0" | "HTTP/1.1"
response-code   = valid response code
header : value  = valid HTTP header and its value
data           = resource state representation (hypertext)
```

Serving HTTP Request




- IP and TCP addressing

1. User enters URL **http://company.cz:8080/orders** to the browser
2. Browser gets an IP address for **company.cz**, **IP:138.232.189.127**
3. Browser and Web Server creates a socket
[147.32.100.5:3223;138.232.189.127:8080]

- Application addressing

4. Browser sends HTTP request, that is, writes following data to the socket
 - 1 | GET /orders HTTP/1.1
 - 2 | Host: company.cz
5. Web server passes the request to the web application **company.cz** which serves **GET orders** and that writes a response back to the socket.

HTTP Listener

- HTTP listener implementation in Java using Jetty 
 - Server listens on port **8080**
 - Jetty parses HTTP request data into **HttpServletRequest** object.
 - When a client connects, the method **handleRequest** is called
 - The method tests the value of the **host** header and responds back if the header matches **company.cz** value.

```
1  /** handles the request when client connects */
2  public void handleRequest(HttpServletRequest request,
3                          HttpServletResponse response) throws IOException, ServletException {
4
5      // test if the host is company.cz
6      if (request.getHeader("Host").equals("company.cz")) {
7          response.setStatus(200);
8          response.setHeader("Content-Type", "text/plain");
9          response.getWriter().write("This is the response");
10         response.flushBuffer();
11     } else
12         response.sendError(400); // bad request
13 }
```

HTTP Listener (Cont.)

- Test it using Telnet

```
1  telnet 127.0.0.1 8080
2  # ...lines omitted due to brevity
3  GET /orders HTTP/1.1
4  Host: company.cz
5
6  HTTP/1.1 201 OK
7  Content-Type: plain/text
8
9  This is the response...
```

- HTTP listener in bash

- Use it to test incoming HTTP connections quickly
- Uses **nc** utility (*netcat*)

```
1  # ctrl-c to stop http listener
2  control_c() {
3      echo -en "\n* Exiting\n"
4      exit $?
5  }
6  trap control_c SIGINT
7
8  for (( ; ; ))
9  do
10     echo -e "\n\n* Listening on port $1..."
11     echo -e "\nHTTP/1.0 204 No Content\n\n" | nc -l $port
12 done
```

Virtual Web Server

- Virtual server
 - Configuration of a named virtual web server
 - Web server uses host request header to distinguish among multiple virtual web servers on a single physical host.
- Apache virtual Web server configuration
 - Two virtual servers hosted on a single physical host

```
1 # all IP addresses will be used for named virtual hosts
2 NameVirtualHost *:80
3
4 <VirtualHost *:80>
5     ServerName company.com
6     ServerAdmin admin@company.com
7     DocumentRoot /var/www/apache/company.com
8 </VirtualHost>
9
10 <VirtualHost *:80>
11     ServerName firm.cz
12     ServerAdmin admin@firm.cz
13     DocumentRoot /var/www/apache/firm.cz
14 </VirtualHost>
```

Better Support for HTTP Testing

- Use **curl** to test HTTP protocol

```
1 Usage: curl [options...] <url>
2
3 -X/--request <command>      Specify request command to use
4 -H/--header <line>          Custom header to pass to server
5 -d/--data <data>            HTTP POST data
6 -b/--cookie <name=string/file> Cookie string or file to read cookies from
7 -v/--verbose                 Make the operation more talkative
```

- Example

```
1 curl -v -H "Host: company.cz" 127.0.0.1:8080
2
3 * About to connect() to 127.0.0.1 port 8080
4 * Trying 127.0.0.1... connected
5 * Connected to 127.0.0.1 port 8080
6 > GET / HTTP/1.1
7 > User-Agent: curl/7.20.0 (i386-apple-darwin10.3.2) libcurl/7.20.0 OpenSSL/0.9.8
8 > Accept: */*
9 > Host: company.cz
10 >
11 < HTTP/1.1 201 OK
12 < Connection: keep-alive
13 < Content-Type: plain/text
14 <
15 < This is the response...
```

Overview

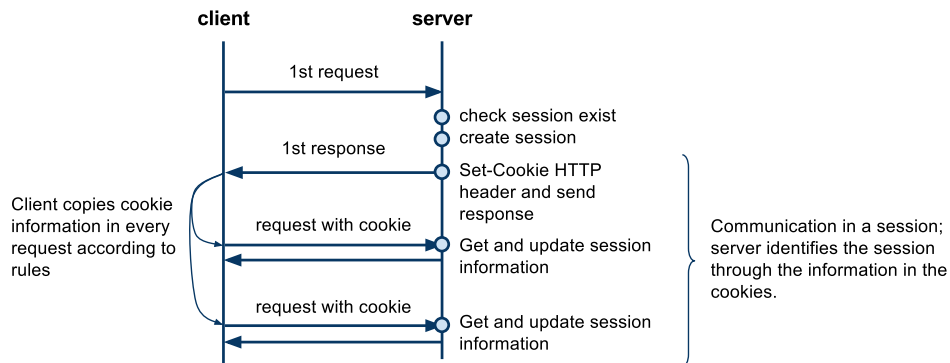
- Introduction to Application Protocols
- Introduction to HTTP
 - *State Management*

State Management

- HTTP is a stateless protocol – original design
 - *No information to relate multiple interactions at server-side*
 - Except **Authorization** header is copied in every request
 - IP addresses do not work, one public IP can be shared by multiple clients
- Solutions to check for a valid state at server-side
 - **Cookies** – obvious and the most common workaround
 - RFC 2109 – HTTP State Management Mechanism [🔗](#)
 - Allow clients and servers to talk in a context called **sessions**
 - **Hypertext** – original HTTP design principle
 - App states represented by resources (hypermedia), links define transitions between states
 - Adopted by the REST principle **statelessness**

Interaction with Cookies

- Request-response interaction with cookies
 - *Session is a logical channel maintained by the server*



- Stateful Server
 - *Server remembers the session information in a server memory*
 - *Server memory is a non-persistent storage, when server restarts the memory content is lost!*

Set-Cookie and Cookie Headers

- **Set-Cookie** response header

```
1 set-cookie = "Set-Cookie:" cookie (";" cookie)*
2 cookie    = NAME "=" VALUE (";" cookie-av)*
3 cookie-av = "Comment" "=" value
4           | "Domain" "=" value
5           | "Max-Age" "=" value
6           | "Path" "=" value
```

- **domain** – a domain for which the cookie is applied
- **Max-Age** – number of seconds the cookie is valid
- **Path** – URL path for which the cookie is applied

- **Cookie** request header. A client sends the cookie in a request if:
 - **domain** matches the origin server's fully-qualified host name
 - **path** matches a prefix of the request-URI
 - **Max-Age** has not expired

```
1 cookie = "Cookie:" cookie-value (";" cookie-value)*
2 cookie-value = NAME "=" VALUE [";" path] [";" domain]
3 path        = "$Path" "=" value
4 domain      = "$Domain" "=" value
```

- **domain**, and **path** are values from corresponding attributes of the **Set-Cookie** header

Session Management Java Class

- Manages client sessions in a server memory 

```
1 public class Sessions<E> {
2
3     // storage for the session data;
4     private Hashtable<String, E> sessions = new Hashtable<String, E>();
5
6     /** Returns session id based on the information in the http request */
7     public String getSessionID(HttpServletRequest request) throws Exception {
8         String sid = null;
9
10
11         // extract the session id from the cookie
12         if (request.getHeader("cookie") != null) {
13             Pattern p = Pattern.compile(".*session-id=([a-zA-Z0-9]+).*");
14             Matcher m = p.matcher(request.getHeader("cookie"));
15             if (m.matches()) sid = m.group(1);
16         }
17
18         // create the session id md5 hash; use random number to generate a client-id
19         // note that this is a simple solution but not very reliable
20         if (sid == null || sessions.get(sid) == null) {
21             MessageDigest md = MessageDigest.getInstance("MD5");
22             md.update(new String(request.getRemoteAddr() +
23                                 Math.floor(Math.random()*1000)).getBytes());
24             sid = Utils.toHexString(md.digest());
25         }
26         return sid;
27     }
28
29     public E getData(String sid) ... // returns session data from sessions object
30     public void setData(String sid, E d) ... // sets session data to sessions object
31 }
```

Stateful Server Implementation

- Simple per-client counter 

```
1 public void handleRequest(HttpServletRequest request,
2                             HttpServletResponse response) throws Exception {
3     // get the session id
4     String sid = sessions.getSessionID(request);
5
6     // create the new data if none exists
7     if (sessions.getData(sid) != null)
8         sessions.setData(sid,
9                             Integer.valueOf(sessions.getData(sid).intValue() + 1));
10    else
11        sessions.setData(sid, Integer.valueOf(1));
12
13    // send the response
14    response.setStatus(200);
15    response.setHeader("Set-Cookie", "session-id="+ sid + "; MaxAge=3600");
16    response.setHeader("Content-Type", "text/plain");
17    response.getWriter().write("Number of hits from you: " +
18                                sessions.getData(sid).toString());
19    response.flushBuffer();
20 }
```



Task

- What happens when the server restarts?
- How do you change the code to count requests from all clients?

Testing

- Testing

- `curl` will require you to specify cookies in every request
- Browser handles cookies automatically

```
1  # run curl for the first time
2  curl -v 127.0.0.1:8080
3  > GET / HTTP/1.1
4  > Host: 127.0.0.1:8080
5  >
6  < HTTP/1.1 200 OK
7  < Set-Cookie: session-id=3a9c3cdc5ff36434aa1ba860727ca401;max-age=3600
8  <
9  Number of hits from you: 1
10
11 # copy the cookie session-id from previous response
12 curl -v -b session-id=3a9c3cdc5ff36434aa1ba860727ca401 127.0.0.1:8080
13 > GET / HTTP/1.1
14 > Host: 127.0.0.1:9900
15 > Cookie: session-id=3a9c3cdc5ff36434aa1ba860727ca401
16 >
17 < HTTP/1.1 200 OK
18 < Set-Cookie: session-id=3a9c3cdc5ff36434aa1ba860727ca401;max-age=3600
19 <
20 Number of hits from you: 2
```