Quiz 2

Montag, 23. Oktober 2017 13:17

1.

a)

The Request-Line begins with a method token, followed by the Request-URI and the protocol version, and ending with CRLF. The elements are separated by SP characters. No CR or LF is allowed except in the final CRLF sequence.

```
    Request-Line   = Method SP Request-URI SP HTTP-Version CRLF
```

Aus <https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html>

b)

```
$ printf 'GET / HTTP/1.1\r\nHost: www.google.com:80\r\n\r\n'
GET / HTTP/1.1
Host: www.google.com:80
```

=>

GET / HTTP/1.1 CRLF Host: 192.168.1.1:8080 CRLF CRLF

c)

stateless: possible by sending all needed information with every request, e.g. within cookies.
client-server: inherently possible with HTTP
cacheable: possible by not changing the system too much while it's live
uniform interface: implementable, we could even say that HTTP headers **are** a uniform-interface. or we could argue about content-type allowing this.
layered system: implementable by chaining services before responding to a request
code on demand: implementable with HTTP body

2.

a)

Socket and ServerSocket.
Socket allows to establish a connection from the Client to a Server, and ServerSocket listens on its port on the server and when a new client connects, it gives the server a Socket that maintains the connection to the Client.

b)

"The underlying data source for some implementations of `InputStream` can signal that the end of the stream has been reached, and no more data will be sent. Until this signal is received, read operations on such a stream can block.
For example, an `InputStream` from a `Socket` socket will block, rather than returning EOF, until a TCP packet with the FIN flag set is received. When EOF is received from such a stream, you can be assured that all data sent on that socket has been reliably received, and you won't be able to read any more data. (If a blocking read results in an exception, on the other hand, some data may have been lost.)"

Aus <https://stackoverflow.com/questions/611760/java-inputstream-blocking-read>

InputStream is blocking on read, that means that the call only returns once the Stream has been fully read. Only then will the program continue.
OutputStream on the other hand will not block if the message to send fits into its buffer. The system providing the Socket will handle the sending of the data asynchronously when the client is ready to recieve it. If the message to send does not fit into the buffer, it is neccessary to flush it - that is to send the buffers content to the client, so that we can refill the buffer.
Flushing does block, because other threads may be using the network interface as well.

In InputStreams, `read()` calls are said to be "blocking" method calls. That means that if no data is available at the time of the method call, the method will wait for data to be made available.
The `available()` method tells you how many bytes can be read until the `read()` call will block the execution flow of your program. On most of the input streams, all call to `read()` are blocking, that's why available returns 0 by default.
However, on some streams (such as `BufferedInputStream`, that have an internal buffer), some bytes are read and kept in memory, so you can read them without blocking the program flow. In this case, the `available()` method tells you how many bytes are kept in the buffer.

Aus <https://stackoverflow.com/questions/3695372/what-does-inputstream-available-do-in-java>

3.

a) REST is not a protocol. **false**
b) **false**, stateless means that the server does not store context
c) **true**, but both POST and GET are a way to send data. However, GET is intended for requesting data while POST is intended to send data, which would allow for arguing that GET corresponds to READ and POST to UPDATE. PUT to CREATE and DELETE to DELETE.
d) According to some sources, JSON is not defined as hypermedia. Wikipedia's stance on this is

"Web resources" were first defined on the World Wide Web as documents or files identified by their URLs, but today they have a much more generic and abstract definition encompassing every thing or entity that can be identified, named, addressed or handled, in any way whatsoever, on the Web. In a RESTful Web service, requests made to a resource's URI will elicit a response that may be in XML, HTML, JSON or some other defined format. The response may confirm that some alteration has been made to the stored resource, and it may provide hypertext links to other related resources or collections of resources. Using HTTP, as is most common, the kind of operations available include those predefined by the HTTP methods GET, POST, PUT, DELETE and so on.

Aus <https://en.wikipedia.org/wiki/Representational_state_transfer>

so I'd say **true**.

4.

a) the WSDL specifications. The URI they can be obtained from needs to be known in advance. E.g for task 2, this was
http://vslab.inf.ethz.ch:8080/SunSPOTWebServices/SunSPOTWebservice?wsdl
This document can be retrieved by requesting it - usually from the lookup service.

b) The most convenient way is via the provided http interface found under http://vslab.inf.ethz.ch:8080/SunSPOTWebServices/SunSPOTWebservice?Tester and directly looking at what it took and will return. However, these element type definitions are referenced and imported in the WSDL file and can be found in the source http://vslab.inf.ethz.ch:8080/SunSPOTWebServices/SunSPOTWebservice?xsd=1 . The element defintions are as follows:

```
<xs:element name="getSpot" type="tns:getSpot"/>
<xs:element name="getSpotResponse" type="tns:getSpotResponse"/>
```

And the complex Types of them:

```
<xs:complexType name="getSpot">
<xs:sequence>
<xs:element name="id" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="getSpotResponse">
<xs:sequence>
```

```
<xs:element name="return" type="tns:sunSpot" minOccurs="0"/>
</xs:sequence>
```

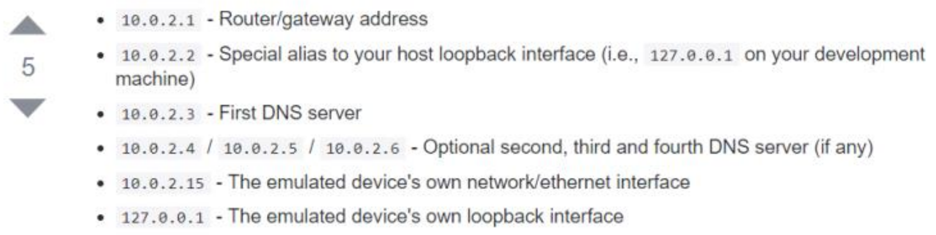
c)
We would probably have to change the port binding:

```
<binding name="SunSPOTWebservicePortBinding" type="tns:SunSPOTWebservice">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
```

to a transport schema defining SMTP
Then, we would probably also need ot update all the URI to the corresponding protocol - replacing http:// with smtp://, including the soap:address.
However, nobody seems to use SMTP instead of HTTP for this, so I as unable to find useful documentation.

5.

a)

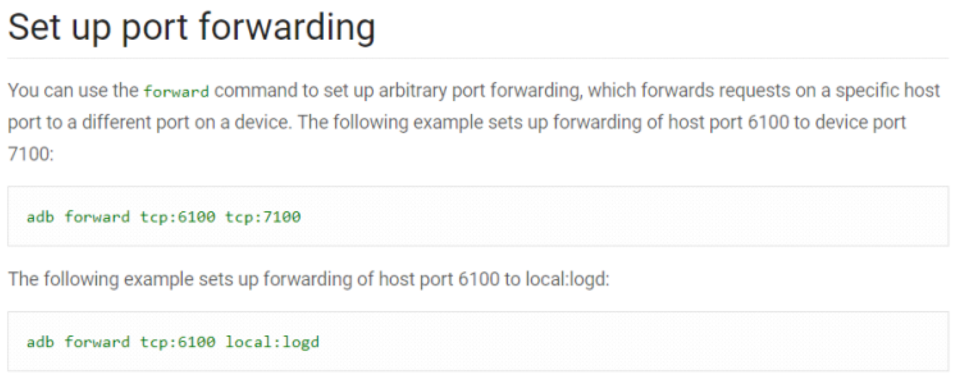

- `10.0.2.1` - Router/gateway address
- `10.0.2.2` - Special alias to your host loopback interface (i.e., `127.0.0.1` on your development machine)
- `10.0.2.3` - First DNS server
- `10.0.2.4` / `10.0.2.5` / `10.0.2.6` - Optional second, third and fourth DNS server (if any)
- `10.0.2.15` - The emulated device's own network/ethernet interface
- `127.0.0.1` - The emulated device's own loopback interface

https://stackoverflow.com/a/16034000/2550406
The device is hidden behind a software router, so every emulated device is in its own subset and thus does not need a different IP.
This probably means, that the emulated devices cannot communicate with each other.

b)
The emulated device itself - it's the loopback interface

c)
see (a). 10.0.2.2

d)


Set up port forwarding

You can use the `forward` command to set up arbitrary port forwarding, which forwards requests on a specific host port to a different port on a device. The following example sets up forwarding of host port 6100 to device port 7100:

```
adb forward tcp:6100 tcp:7100
```

The following example sets up forwarding of host port 6100 to local:logd:

```
adb forward tcp:6100 local:logd
```

https://developer.android.com/studio/command-line/adb.html#forwardports
https://stackoverflow.com/a/4738690/2550406
We can set up a port forward between the devices and some ports of the host machine to allow them to communicate. If we forward port 8080 of the emulator to port 9090 of the host, the host can access it via localhost:9090.