UNIVERSITY OF COPENHAGEN

Communication in Java using Jetty

Vivek Shah <bonii@diku.dk>
Frederik M. Madsen <fmma@diku.dk>

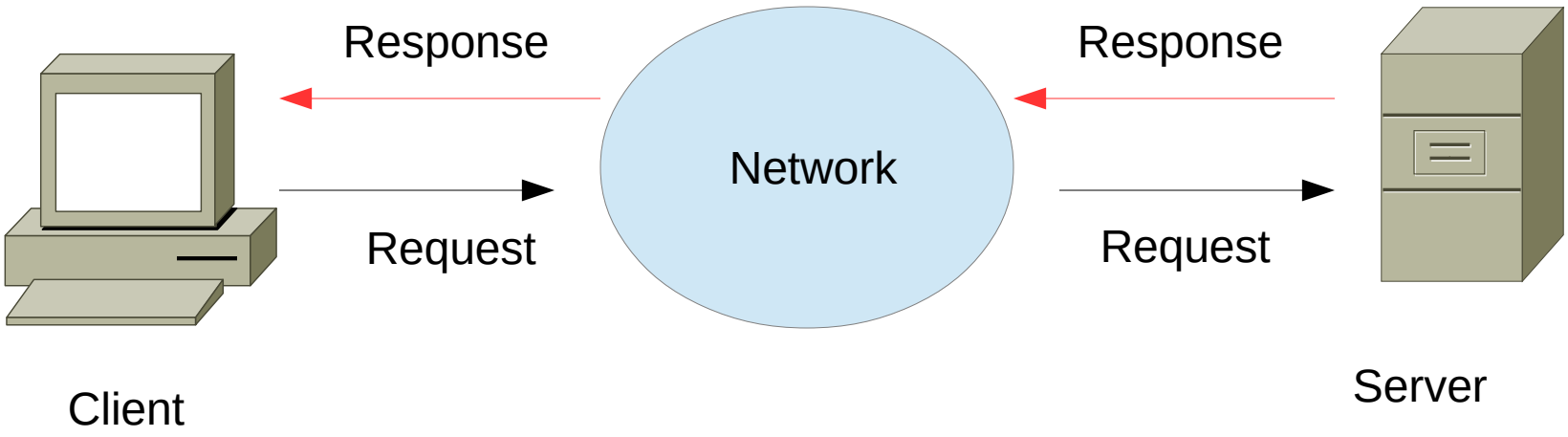
DIKU

Marcos Vaz Salles <vmarcos@diku.dk>
Course Responsible

Tentative Plan

- Day 3: Using Jetty for communication
  - Request Response Model.
  - Using HTTP for communication.
  - Embedding Jetty.(Using Jetty 8)
  - Starting a server.
  - Sending and receiving HTTP messages.
  - Other Jetty features.

Request Response Model



Request Response Model

Server responsibilities

- Needs to listen for client requests.
- On receipt of client request, process the request.
- Send a response back to the client when the processing completes.

Client responsibilities

- Generate and send the request.
- Be prepared for a response.

HTTP and request response model

Hyper Text Transfer Protocol is

- An application protocol which implements the request response client server model.
- Uses TCP (transport layer) for reliable message delivery on an IP network.
- Messages sent in HTTP request and response messages.
- HTTP is stateless.
- Designed to allow scalability in intermediate hardware and software components (proxies).
- Use HTTP for communication instead of building messaging protocol.

HTTP requests

- HTTP requests are specified using URLs.

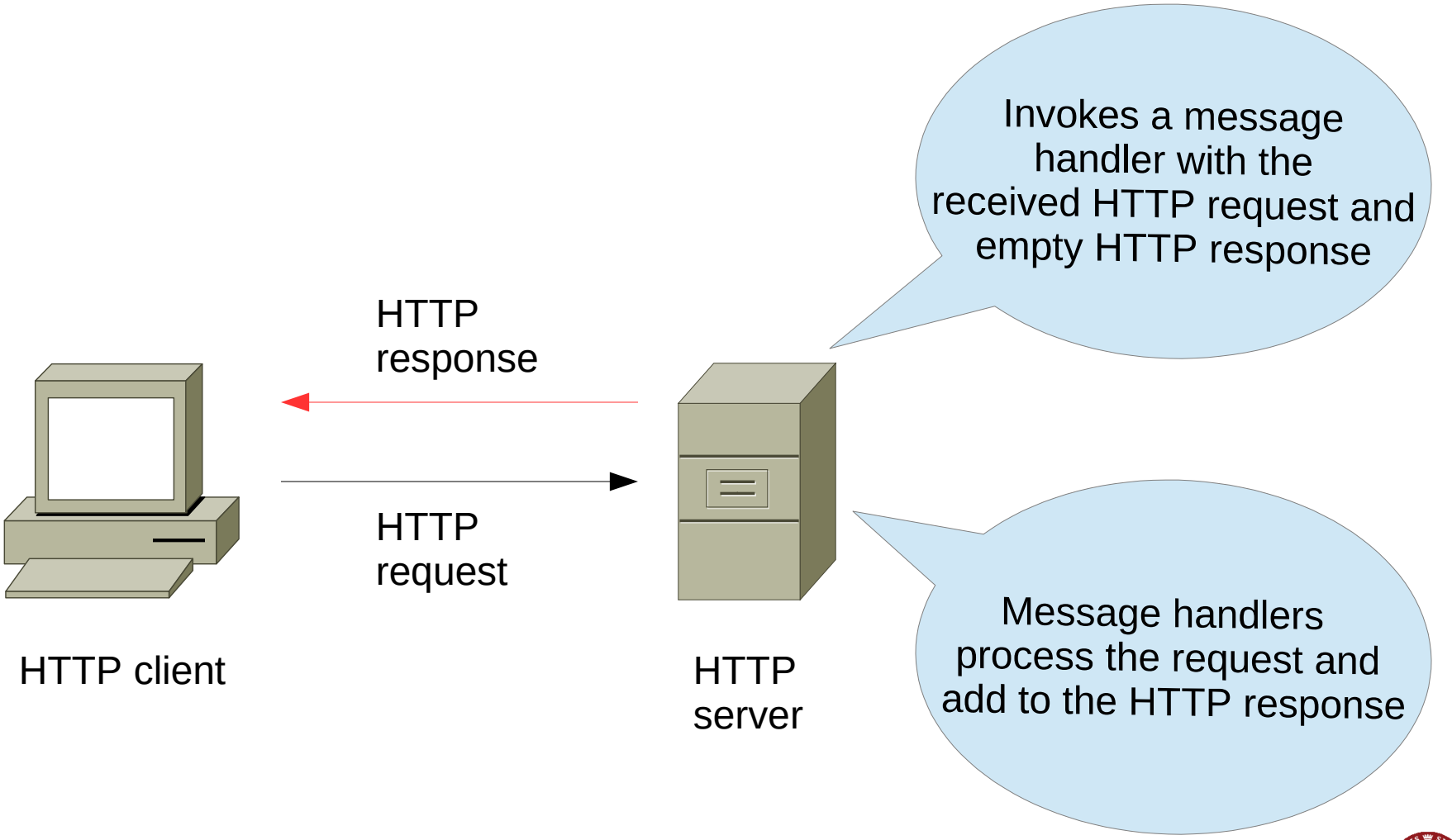
  URL = domain:port/path?query_string

- http://www.google.co.dk/search?
  q=search+engine+optimisation&ie=utf-8

  query_string : field1=value1&field2=value2&.....

- HTTP request methods
  - GET
  - HEAD
  - POST
  - OPTIONS

HTTP communication workflow



Example HTTP requests and response

```
GET /index.html?query=foo HTTP/1.1
Host: www.example.com
```

HTTP request

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
ETag: "3f80f-1b6-3e1cb03b"
Content-Type: text/html; charset=UTF-8
Content-Length: 131
Connection: close
<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML
document.
</body>
</html>
```

HTTP response

To use HTTP for communication

- Need to implement HTTP request methods on clients.
- Need to implement an HTTP server
- Need to implement HTTP response methods on server.
- Need to build delegation of processing to application handlers.
- Handle response methods on clients.
- Thread pool management.
- Network stack management.
- And a lot of everything else for performance.

Introducing Jetty

- Jetty provides HTTP client libraries to enable HTTP requests and wait for HTTP responses.
- Jetty provides HTTP server which listens for HTTP requests and invokes registered handlers to handle requests.
- Jetty provides all the necessary tools (client and server libraries) to enable efficient use of HTTP for communication.
- In reality, Jetty is an application server !!
- We will deploy Jetty inside our applications and not the other way around.

Embedding Jetty inside application

- Create a server instance.
- Add/configure connectors.
- Add/configure handlers.
- Start the server.
- Wait on the server or do something else with your thread.

Starting the server

```
public class SimpleServer
{
    public static void main(String[] args) throws Exception
    {
        Server server = new Server(8080);
        server.start();
        server.join();
    }
}
```

Starting the server

Lets create one

Connectors

- Connectors are links of the server to the outside world which supports the HTTP protocol.
- Each connector element represents a port that the Jetty server will listen on.
- The connectors have just one job → To listen for messages and pass them to the core engine.
- Using connectors, we can control how the HTTP server listens for messages.
- A default connector is created when a server starts if no connectors are specified.
- One server can have multiple connectors.

Connectors

- Connectors allow us to control
  - ServerHost
  - ServerPort
  - IdleTimeout
  - MaxRequestSize
  - MaxRequestParameters
  - MaxResponseSize
  - ThreadPoolSize
  - Security scheme (one can configure a HTTPS connector)
- In Jetty, create connectors using SelectChannelConnector class.

Connectors

```
public class SimpleConnector {
    public static void main(String[] args) throws Exception {
        Server server = new Server();
        ServerConnector http = new ServerConnector(server);
        http.setHost("localhost");
        http.setPort(8080);
        http.setIdleTimeout(30000);
        server.addConnector(http);
        server.start();
        server.join();
    }
}
```

Request Handlers

In order to handle incoming requests, Jetty requires registration of request handlers. A handler may

- Examine/modify HTTP requests.
- Generate the complete HTTP response.
- Call another handler.
- Select one or many handlers for invocation.
- Create handler by extending AbstractHandler class
- Implement the handle method. (remember it can be executed by multiple threads)

Request Handler

```
public class SimpleHandler extends AbstractHandler {

    public void handle(String target, Request baseRequest,
        HttpServletRequest request,HttpServletResponse
        response) throws IOException, ServletException {

         response.setContentType("text/html;charset=utf-8");

         response.setStatus(HttpServletResponse.SC_OK);

        response.getWriter().println("<h1>Hello World</h1>");

        baseRequest.setHandled(true);
    }

}
```

Request Handler

Lets see it in action

Parameters to request handler

The parameters passed to the handle method are:

- String target (arg0) – the target of the request, which is either a URI or a name from a named dispatcher.
- Request baseRequest (arg1) – the Jetty mutable request object.
- HttpServletRequest request (arg2) – the immutable request object.
- HttpServletResponse response (arg3)– the response object
- The handler sets the response status, content-type, and marks the request as handled before it generates the body of the response using a writer.

Sophisticated request handlers

- A Handler Collection holds a collection of other handlers and calls each handler in order.
- A Handler List is a Handler Collection that calls each handler in turn until either an exception is thrown or request.isHandled() returns true.
- A Handler Wrapper is a handler base class that can be used to daisy chain handlers together in the style of aspect-oriented programming.
- And you have servlets (that is for another day).

Passing arguments in HTTP request

http://localhost:8080/increment

http://localhost:8080/decrement

http://localhost:8080/getcount

http://localhost:8080/addby?value=10

http://localhost:8080/counter?type=increment

http://localhost:8080/counter?type=decrement

http://localhost:8080/counter?type=getcount

http://localhost:8080/counter?type=addby&value=10

Retrieving arguments

- getParameter(“parameter-name”) is available in HTTPServlet class to retrieve URL parameter.
- getParameterMap() is available in HTTPServlet class to retrieve “all” parameters and their values.
- Jetty supports duplicate parameters with different values, part of HTTPServlet specification but not HTTP specification.
- Encode and decode parameter values and parameter names using URLEncoder and URLDecoder.
- Be careful to encode and decode your query strings (if needed)!

Do we need POST requests?

- Activity: Discuss in groups for 3 minutes if “GET” requests suffice for all scenarios that a “POST” request can be used for.
- One can send complicated data-structures using “GET” encodes in the query string.
  - http://localhost:8080/runProgram?input=a&input=b&input=c
  - http://localhost:8080/runProgram?n=2&input1=a&input2=b
- Duplicate parameters are supported by “HTTPServlet” specification but not HTTP specification.
- Really hard to structure/debug. What goes around comes around. Do not cut short your lifetime on Earth.
- What about binary data ?

Accessing data from POST requests

```
public String extractPost(HttpServletRequest req)
{
    int len = req.getContentLength();
    BufferedReader reqReader = req.getReader();
    char[] cbuf = new char[len];
    reqReader.read(cbuf);
    reqReader.close();
    return new String(cbuf);
}
```

- Generic mechanism to read character encoded data.
- You can use req.getInputStream to access binary data as byte stream.

Accessing data from POST requests

- Once you read from the request, the read content is removed from the request.
- Make sure you close the reader to avoid leaks.
- getParameter is supported in POST requests as well only if the client encodes the data as parameters(form-url-encoded).
- Better to use the generic method for POST.

Building clients (HttpClient)

- We used browser as a client → Enabled us to test server side methods.
- Let us build a client now (using HTTP for communication remember).
- HttpClient is the Jetty component that allows us to make requests and interpret responses to HTTP servers.
- HttpClient is asynchronous → supports callbacks.
- HttpExchange represents a request-response exchange with the HTTP server.

Starting an HTTPClient

```
client = new HttpClient();
client.setConnectorType(HttpClient.CONNECTOR_SEL
ECT_CHANNEL);
client.setMaxConnectionsPerAddress(300);
client.setThreadPool(new QueuedThreadPool(20));
client.setTimeout(30000);
client.start();
```

- Client can be viewed as a multi-threaded service.

- Once a client is setup, exchanges may be performed HttpClient.send(HttpExchange exchange) method.

Using HTTPClient

- The design of HTTPClient allows concurrent thread-safe exchanges.
- You want to re-use the same client for all HTTP communication, you would not want to start a new client for each HTTP exchange.
- Invoking stop() method on the client stops it.

Exchanging synchronous messages with server (GET request)

```
public static void sendGetRequest() throws Exception {
    ContentExchange exchange = new ContentExchange();

    exchange.setURL("http://localhost:8080/foo?time=a");

    client.send(exchange);
    int exchangeState = exchange.waitForDone();

    if (exchangeState == HttpExchange.STATUS_COMPLETED) {
        System.out.println(exchange.getResponseContent());
    } else {
        System.out.println(“Error occured”);
    }
}
```

Exchanging synchronous messages with server (GET request)

- An exchange represents a request-response workflow.
- ContentExchange is a subclass of Exchange. Useful wrappers to access request and response content.
- By default all exchanges use “GET” request.
- Client.send(exchange) is asynchronous.
- `getResponseContent()` - Retrieves the result (String).
- Exchange provides fine grained control to invoke callbacks at various points in the HTTP protocol.
- For this tutorial, we are only using synchronous exchanges.

Exchanging synchronous messages with server (POST request)

```
import org.eclipse.jetty.io.Buffer;
import org.eclipse.jetty.io.ByteArrayBuffer;

public static void sendGetRequest() throws Exception {
    ContentExchange exchange = new ContentExchange();
    exchange.setMethod(“POST”);

    exchange.setURL("http://localhost:8080/foo?time=a");
    Buffer postData = new ByteArrayBuffer("post data");
    exchange.setRequestContent(postData);

    client.send(exchange);
    int exchangeState = exchange.waitForDone();

    if (exchangeState == HttpExchange.STATUS_COMPLETED) {
        System.out.println(exchange.getResponseContent());
    } else {
        System.out.println(“Error occured”);
    }
}
```

Exchanging synchronous messages with server (POST request)

- Use Buffer to create the “POST” data content.
- setResponseContent(buffer) → Sets the buffer as the contents of the request.
- Equivalent classes for byte streams exist as well. For this tutorial, lets just consider text data.

And Jetty has lots more. Explore Jetty at http://www.eclipse.org/jetty/

The need for serialization and de-serialization

- How do we send data-structures as text ?
- We need to serialize and de-serialize.
- Possible representations → XML, JSON ….
- For this course we will use XML → Xtream library.
- We can also serialize and de-serialize to binary format.
- Using Java serialization, protocol buffers, other libraries.
- Will text based serialization/de-serialization hamper us ?

Using Xstream(let's look at code)

Xstream has lots more

Explore Xstream

http://xstream.codehaus.org/

Advantages of using HTTP

- Connect our applications with other HTTP servers. Allows layering.
- Leverage the performance of HTTP servers.
- Use proxy servers for free. Increase fan out.
- Leverage HTTP mechanisms like Caching (etags, ttl).
- Use system stack of the internet.
- Allows us to leverage existing libraries without sacrificing performance.
- Most often we end up building some form of reliable message delivery protocol using sockets, why not use HTTP instead ?

Limitations of HTTP as communication

- Latency sensitive applications.
- Limitations of TCP.
- Request/response model does not fit communication pattern in all applications
    - Stream based systems
    - VOIP
    - Online games.