6. HTTP

6.1 Learning Objectives

In the previous chapter, I unfolded the *Broker* pattern as well as an implementation of it for the TeleMed and GameLobby systems. In the IPC layer, I used Java sockets, which is Java's fundamental network implementation. In this chapter, I will focus on another IPC approach that has achieved enormous success, namely the *hypertext transfer protocol* (HTTP). As the main engine of the world wide web, it is popular and is supported by numerous high quality frameworks, that I can reuse in implementing a strong IPC layer.

In this chapter, I will start by outlining the basic concepts in HTTP, and once this foundation has been established, I will move on to discuss how *Broker*'s IPC layer can easily be implemented using it.

6.2 A HTTP Walk-through

HTTP is the application protocol that powers the World Wide Web (WWW). With the huge success of WWW for human information search and retrieval, a lot of high quality software frameworks for building WWW client and server software became available. In addition, the HTTP protocol has a number of interesting properties that allows scaling, that is, handle large workloads and traffic. Combined, HTTP is an ideal platform for building distributed computing systems.

But, let us first take a quick look at the HTTP protocol. The presentation here is by no means comprehensive, but should provide enough information for understanding it, our implementations of the Broker pattern on top of it, and the next chapter on REST.

HTTP is a standard for the request-reply protocol between web browsers (clients) and web servers in the standard client-server architecture. The client sends requests to the webserver for a web page and the server sends a response back, typically an HTML formatted page.

Message Format

All HTTP messages (requests and responses) are simple plain-text messages. The format of a request has the following format:

- A request line, that specifies the HTTP verb, the wanted resource, and the version of the HTTP protocol.
- Request header fields, which are key-value pairs defined by the HTTP protocol.
- An empty line.
- An optional message body.

As an example, if you request your browser to fetch the URL

```
http://www.baerbak.com/contact.html
```

the following text message will be sent by the browser to my book's web server, whose address on the internet is "www.baerbak.com":

```
 GET /contact.html HTTP/1.1
 Host: www.baerbak.com
 Accept: text/html
```

Try it out, for instance using *curl*, see Sidebar: Curl. If you do, then you have requested my book's web server to return the contents of the "contact.html" page.

Curl

As HTTP is a plain-text format, you can actually quite easily communicate with web servers right from the shell. The tool `curl` is handy.

Try the following command which will output both the request message as well as the response message (-v).

```
curl -v www.baerbak.com/contact.html
```

GET is one of the HTTP verbs, discussed in the next section, and this one simply requests the resource mentioned: the page "contacts.html". The headers "Host" and "Accept" defines the host name of the web server (www.baerbak.com) and what return types (text/html) are accepted respectively.

In the reply, the web server will return the requested resource in a response message, that is similar to the request format:

- A single status line which includes the status code and status message.

- Response header fields, again a set of well-defined key-value pairs, on multiple lines.
- An empty line.
- The (optional) message body, that is the contents of the requested resource.

The response to the above request is shown here:

```
HTTP/1.1 200 OK
Date: Mon, 19 Jun 2017 09:58:25 GMT
Server: Apache/2.2.17 (FreeBSD) mod_ssl/2.2.17 OpenSSL/1.0.0c ...
Last-Modified: Mon, 13 Apr 2015 12:34:07 GMT
ETag: "b46bce-676-5139a547e2dc0"
Accept-Ranges: bytes
Content-Length: 1654
Vary: Accept-Encoding,User-Agent
Content-Type: text/html

<html>
  <head>
    <title>Flexible, Reliable Software</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <link href="style.css" rel="stylesheet" type="text/css">
...
```

The status line returned the 200 HTTP status code which is “OK” — everything works as expected. HTTP defines a rich set of status code that tell clients specifically what went wrong in case of errors, and also indicate how to remedy the situation. They are discussed in more detail in Section Status codes.

Next follows a series of header fields, one notable one is “Content-Type” that states the format of the returned resource: “text/html” which tells the receiver that it is text formatted in the HyperText Markup Language (HTML).

Finally, the response contents follows after the empty line in the returned message. As stated by the “Content-Type” the returned contents is formatted using HTML which allows a web browser to display the contents in a visually and typographical correct way.

Uniform Resource Identifier / URI

Clients and servers communicate about *resources*, like our web page in the example above. A resource is **information that can be named**, which calls for some uniform way of defining a name. **Uniform Resource Identifier (URI)** is a string based schema for how to name resources. The full schema is

```
scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]
```

However, I will only use a simple subset of this schema, namely

```
scheme:[//host[:port]][/path]
```

Uniform Resource Locators (URL) follows the URI schema and identifies both the resource's *location* as well as the *means for retrieving* it.

One such example resource is the 'contact' homepage retrieved above at *http://www.baerbak.com/contact.html*. It states that the resource "contact.html" can be found on the web server "www.baerbak.com" and the means to retrieve it is the HTTP protocol.

HTTP Verbs

The HTTP version 1.1 protocol includes a number of **verbs**. The most widely used when browsing is the GET verb which simply means "get that resource for me", but there are others. Those relevant for our discussion are:

Verb	Action	Idempotent?
GET	request representation of a resource (URI)	Yes
POST	accept enclosed entity as new subordinate of resource (URI)	No
PUT	request enclosed entity to be stored under URI	Yes
DELETE	request deletion of resource (URI)	Yes

In contrast to the GET verb which simply retrieves existing information from a web server, the three other verbs allow clients to create (POST), modify (PUT), and delete (DELETE) information on a web server. Of course, such modifications of contents on the web server is only possible if the given web server accepts these verbs and the request is well formed.

All verbs, except POST, are *idempotent* which means that executing them several times has the same effect as only executing it once. If you send one or ten identical PUT requests to the server, the outcome is the same, namely that the named resource is updated. The POST is different, as sending two POST requests will create two resources, etc.

In the database community, you often speak of the **CRUD** operations: **C**reate, **R**ead, **U**pdate, and **D**elete; which are the four basic operations that a database supports. If you compare the HTTP verbs with CRUD, the are identical except for the naming:

- Create: is the POST verb

- Read: is the GET verb
- Update: is the PUT verb
- Delete: is the DELETE verb

Thus, a web server supporting all four HTTP verbs is fundamentally a database of resources that can be created, read, updated, and deleted.

HTTP Status Codes

The HTTP protocol also defines a well defined vocabulary of **status codes** that describe outcomes of any operation whether they succeed or fail. The status codes are numerical values, like 200, and they are grouped into response classes, so all responses in range 200 - 299 are 'Success' codes, all in range 500 - 599 are server errors, etc.

Some codes are much more used than others, so I will focus on these

- **200 OK**: Standard response for successful HTTP requests. Typically GET requests that were served correctly return 200 OK.
- **201 Created**: The request has been fulfilled, resulting in the creation of a new resource. This is the status code for a successful POST request.
- **404 Not Found**: The requested resource could not be found but may be available in the future. For instance if you GET a resource that is not present on the server, you get a 404.
- **501 Not Implemented**: The server either does not recognize the request method, or it lacks the ability to fulfill the request.

You can find the full list of status codes on Wikipedia.

Media type

The client and server need to agree on the data format of the contents exchanged between them. Sending JSON to a server that only understands XML of course leads to trouble.

Therefore a central internet authority, Internet Assigned Numbers Authority (IANA), maintains a standard of possible *media types*, that is, standard formats. The media type is essential for a web browser in order to render read data correctly. For instance, the media type *text/html* states that the returned contents is HTML which the browser can then render correctly, if on the other hand the media type is *text/plain* it is just plain text that needs no further processing. Similar there are media types for images like *image/jpeg* or *image/gif*. Generally the format is in two parts that state *type/subtype*.

As our discussion is about programmatic exchange of data between server and client, the type *application* is most interesting, and the media types for XML: *application/xml* and JSON: *application/json* are the ones you will encounter.

Media types were formerly known as MIME types or content types.

For a client and server to express the media type of the contents of a GET, PUT or POST message you use the HTTP headers.

To *request* that contents is formatted according to a specific media type, you set the 'Accept' header in the request message, like

```
1  GET /contact.html HTTP/1.1
2  Host: www.baerbak.com
3  Accept: text/html
```

This states: "Send me the contents in HTML".

In the *response* message, the server will provide information about the media type of the returned contents, using the 'Content-Type' header:

```
1  HTTP/1.1 200 OK
2  Date: Tue, 15 May 2018 07:33:46 GMT
3  ...
4  Content-Type: text/html
```

In the ideal world, a server will be able to return contents in a variety of formats, like *application/xml* and *application/json*, depending upon what the client requests. However, this of course requires extra effort of programming, and is all too often avoided. If a server cannot return contents in the requested format, the HTTP status code *406 Not Acceptable* should be returned. Again, this is the ideal situation, all too often it just returns data in the format it knows, so be sure to check the 'Content-Type' in the client to be sure to parse correctly.

6.3 A HTTP Case Study: PasteBin

It is easy to try out GET on a normal web server, but to showcase the other verbs, I will show how to write a rudimentary web server that accepts POST and GET messages. As client, I will simply use 'curl'.

Our case, *PasteBin*, is a simple online "database"/clipboard that you can copy a small chunk of text to, using POST, for later retrieval, using GET.

POST to create a resource

Our pastebin web server will accept POST messages encoded in JSON. The JSON format must have a key, *contents*, whose value is a string representing the 'clip' that we want to store on the pastebin server. Or in HTTP terms, we want to *create a resource that names our information*.

Furthermore, our server accepts only clips on the resource path */bin*, which represents bins of JSON contents. Let us hop into it, you can find the source code in the FRDS.Broker Library[1] in the *pastebin* folder.

Start the web server using Gradle

```
gradle pastebin
```

This will start the pastebin server on `localhost:4567/bin`.

Next I want to store the text 'Horse' on the pastebin server, so I send it a POST message using 'curl':

```
curl -i -X POST -d '{"contents":"Horse"}' localhost:4567/bin
HTTP/1.1 201 Created
Date: Mon, 07 May 2018 09:13:58 GMT
Location: localhost:4567/bin/100
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.6.v20170531)

{"contents":"Horse"}
```

OK, let us take a look at each aspect in turn.

The first line is the curl call which states: *Send a POST message (-X POST) to the server at URI localhost:4567/bin, with a message body (-d) that contains '{"contents":"Horse"}', and let me see the full HTTP reply message (-i)*. So, 'curl' will print the message response which follows the reply format.

The status code is 201 Created, telling me that the resource was stored/created. Next follows header fields. The most important one when you POST messages is the Location key/value pair:

```
Location: localhost:4567/bin/100
```

As you create contents when POSTing, the server must tell you where the information is stored ("named"), and this is the *Location* header field, which

[1] https://bitbucket.org/henrikbaerbak/broker

here states that the URL for all future references to this resource is *localhost:4567/bin/100*.

Finally, a web server may actually change the provided information, so it will transmit the named resource back to you. Here, however, no changes have been made, so it just returned the original JSON object, as the message body.

GETing the resource

Now, to retrieve our clip from the server, I use the GET verb on the resource URI that was provided in the Location field.

```
curl -i localhost:4567/bin/100
HTTP/1.1 200 OK
Date: Mon, 07 May 2018 09:15:47 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.6.v20170531)

{"contents":"Horse"}
```

The status code is 200 OK, and the JSON object returned in the message body.

If I try to access a non-existing resource, here bin 999:

```
curl -i localhost:4567/bin/999
HTTP/1.1 404 Not Found
Date: Mon, 07 May 2018 09:16:27 GMT
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(9.4.6.v20170531)

null
```

I get a 404 Not Found, and the message body contains 'null'.

Try to store different text clips in a number of bins and retrieve them again.

Implementing the Server

Implementing a quality web server is not trivial. However, one of the big benefits of the success of world wide web is, that there are numerous open source web server frameworks that we can use.

I have chosen Spark[2] as it has a shallow learning curve, and you need only to write little code to handle the HTTP verbs. It uses *static import* and *lambda functions* introduced in Java 8.

The basic building block of Spark is a set of *routes* which represent *named resources*. Static methods define the HTTP verb, and the parameters define the URI and the lambda function to compute a response, like

```
get("/somepath", (request, response) -> {
    // Return something
});

post("/somepath", (request, response) -> {
    // Create something
});
```

Returning to our PasteBin, POST messages are handled by the following piece of code:

```
post("/bin", (req, res) -> {
    // Convert from JSON into object format
    Bin q = gson.fromJson(req.body(), Bin.class);

    // Create a new resource ID
    String idAsString = ""+id++;

    // Store bin in the database
    db.put(idAsString, q);

    // 201 Created
    res.status(HttpServletResponse.SC_CREATED);
    // Set return type
    res.type("application/json");
    // Location = URL of created resource
    res.header("Location", req.host()+"/bin/"+idAsString);

    // Return the constructed bin
    return gson.toJson(q);
  });
```

The declaration states that a handler for POST messages is defined on the URI path "/bin", followed by the lambda-function defining the handler. The first line of that method converts the request's body (`req.body()`) from JSON into the internal class `Bin` using GSON. (`Bin` is just a class that contains a single

[2] http://sparkjava.com/

String type named 'contents' and thus match the `{"contents":"Horse"}` message I used earlier.) The next two lines just creates a resource id and stores the object in an "database" called 'db'. The result object, `res`, is modified so the status code is 201, and the *Location* field assigned. Finally, the constructed 'bin' is returned which becomes the message body.

GET messages are handled by the following piece of code.

```
1  get("/bin/:id", (req, res) -> {
2      // Extract the bin id from the request
3      String id = req.params(":id");
4
5      // Set return type
6      res.type("application/json");
7
8      // Lookup, and return if found
9      Bin bin = db.get(id);
10     if (bin != null) {
11       return gson.toJson(bin);
12     }
13
14     // Otherwise, return error
15     res.status(HttpServletResponse.SC_NOT_FOUND);
16     return gson.toJson(null);
17   });
```

The algorithm is similar to the POST one. One important feature of Spark is that a path which contains an element starting with a colon, like `:id` above, are available through the `params()` method. So, if I send a GET request on `/bin/100`, then `req.params(":id")` will return the string "100".

Note also that the 200 OK status code is not assigned in the code fragment above, as this is default value assigned by Spark.

Extend the PasteBin systems so it also supports UPDATE and DELETE requests.

6.4 Broker using HTTP

So - we have HTTP: a strong request-reply protocol that enables us to transfer any payload between a client and a server, backed up by high quality open source web server implementations. This is an ideal situation to base our Broker pattern on.

Remember from our previous chapter that we can use any network transport layer as IPC by implementing proper pairs of *ClientRequestHandler* and *ServerRequestHandler*. Their responsibilities are to transport payloads of bytes, the marshalled requests and replies, between the client side and the the server side.

Thus, a basic usage of the WWW infrastructure is simply to program a *ClientRequestHandler* and a *ServerRequestHandler* that use HTTP as transport layer. This usage is termed *URI Tunneling* as we use a single URI as a tunnel for transporting our messages.

You can find the source code in package `frds.broker.ipc.http` in the `broker` project in the FRDS.Broker library[3].

Server Side

A method call may create objects or change the state of objects on the server side and thus the POST verb is the most appropriate to use. Indeed I will be lazy and use POST for *all* methods irrespective of whether they are accessors or mutators - which is quite normal in URI tunneling, though certainly not consistent with the envisioned use of the HTTP verbs.

Finally, the URI path is not that relevant as we just use HTTP as a tunnel for exchanging JSON request and reply objects - it does not name any particular resource, just a tunnel. In TeleMed, this tunnel is on the path "/bp" (short for bloodpressure).

The server request handler is responsible for accepting the POST requests, hand it over to the invoker, and return the reply. The implementation below again uses the Spark framework. The path is assigned elsewhere in the 'tunnelRoute' instance variable to "/bp/".

(Fragment from *broker* package: UriTunnelServerRequestHandler.java):

```
1  // POST is for all incoming requests
2  post(tunnelRoute, (req,res) -> {
3    String marshalledRequest = req.body();
4
5    String reply = invoker.handleRequest(marshalledRequest);
6
7    // The reply is opaque - so we have no real chance of setting a proper
8    // status code.
9    res.status(HttpServletResponse.SC_OK);
10   // Hmm, we also do not know the actual marshalling format but
11   // just know it is textual
12   res.type(MimeMediaType.TEXT_PLAIN);
```

[3] https://bitbucket.com/henrikbaerbak/broker

```
13
14
15    return reply;
16  });
```

Note that I do not need to set the *Location* field when just using a tunnel. Our payload contains all object id's, and there is no need.

Client Side

In PasteBin, I just used 'curl' on the command line to interact with the web-server. However, in TeleMed I of course need a programmatic way to send requests to the server. Here, I am using the UniRest[4] library which allows HTTP requests to be constructed using a fluent API. So the `sendToServerAndAwaitReply()` method will look like this:

(Fragment from *broker* package: UriTunnelClientRequestHandler.java)

```
1  @Override
2  public String sendToServerAndAwaitReply(String request) {
3    HttpResponse<String> reply;
4
5    // All calls are URI tunneled through a POST message
6    try {
7      reply = Unirest.post(baseURL + path)
8              .header("Accept", MimeMediaType.TEXT_PLAIN)
9              .header("Content-Type", MimeMediaType.TEXT_PLAIN)
10             .body(request).asString();
11   } catch (UnirestException e) {
12     throw new IPCException("UniRest POST request failed on request="
13             + request, e);
14   }
15   return reply.getBody();
16 }
```

The core here is line 7-10 which is a single fluent statement. It tells UniRest to issue a POST request, `UniRest.post()`, on the given path. The `baseURL` and `path` variables are set elsewhere, and in the concrete case the concatenation of these variables result in the string "http://localhost:4567/bp", so the request is sent to locally deployed server on the bloodpressure path "/bp", as we defined in the previous section.

Line 8 and 9 define header key/value pairs, and the final method call `body().asString()` in line 10 defines first the body of the request message (`body()`), and finally that the returned reply from the server must be interpreted as String

[4] http://unirest.io/

(`asString()`) as we do not know the actual used marshalling format - they are just opaque byte arrays/strings.

The last part of the method extracts the actual marshalled string from the reply and returns it back to the calling **Requestor**.

6.5 Summary of Key Concepts

The Hyper Text Transfer Protocol (HTTP) is an application level protocol for transferring data between web browers and web servers. It is a text based format using a fixed, line-oriented, structure of both request and response messages. A request message states the HTTP verb (GET, POST, PUT, DELETE, and others), header fields, and optionally a message body containing data. POST and PUT uploads data in the message body to the web server (they are Create and Update messages respectively), while GET and DELETE do not (they are Read and Delete messages). The response contains a status code from a rich vocabulary of codes that details the outcome of the server operation.

HTTP and associated quality web frameworks (like Spark, UniRest, and many others) form an excellent framework for implementing the *Broker* pattern. A widely used technique is URI Tunneling which simply handles all remote method calls on a single tunnel path, all handled by POST messages that contains the full method call payload in the message body.

6.6 Review Questions

Enumerate and explain the four HTTP verbs. Which ones are idempotent and which are not? What does idempotent mean?

Outline and explain the format of HTTP request and reply messages.

Explain the URI scheme for naming resources. Give some examples from ordinary web browsing.

What is a media type and why is it important to state it in requests and replies? Name some media types. Which header keys are used to set/define the media type?

Explain how the HTTP protocol can be used as the *Broker* IPC implementation. Explain why both the creation and reading of TeleMed blood pressure measurements are using POST messages in my implementation, and explain why it does not align with the intention of the HTTP verbs.