Pervasive Data Management, 2014
Politecnico di Milano

HTTP protocol integration in PerLa

Federico Monterisi, Fabio A. Schreiber[1], Emanuele Panigatti [2]

Abstract
In this document it is described the implementation of HTTP protocol in PerLa framework.
After a summary of *what the HTTP protocol is* (Section 1), the subsequent sections (2 and 3) deal with the design of a generic Channel in PerLa and the implementation of HTTP Channel, following REST principles.
In the last section of document (Section 4) it will be illustrated which new features are enabled from HTTP Channel implementation.

[1] *Full Professor, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy*
[2] *PhD student, Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy*

Contents

Introduction

In this document it is described the implementation of HTTP protocol in PerLa framework.
After a summary of *what the HTTP protocol is* (Section 1), the subsequent sections (2 and 3) deal with the design of a generic Channel in PerLa and the implementation of HTTP Channel, following REST principles.

In the last section of document (Section 4) it will be illustrated which new features are enabled from HTTP Channel implementation.

Before understanding how HTTP protocol works in PerLa it is important to understand why to use it.

Why HTTP protocol integration?

In the first versions PerLa network was thinking for connect sensors through TCP/IP protocol[3]. But now some information are retrievable by other system not only by sensors.
Thinking about weather forecast (temperature, pressure), there are numerous Web Services offering APIs (JSON/REST or SOAP). These APIs gives information that PerLa could be model like a virtual sensor.
In this way FPC extends his functionality and, for example, connecting a light sensor with TCP/IP protcol and a weather forecast Web Service with SOAP, this FPC gives information about luminosity, temperature and pressure without physical device installed. The Figure 1 shows a possible configuration of two FPC in PerLa.

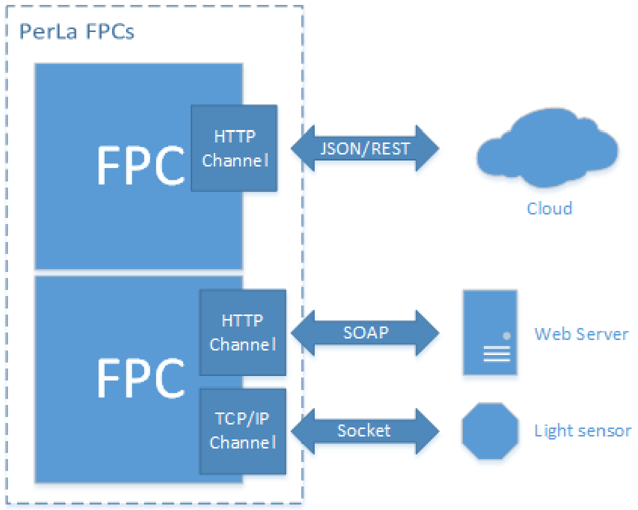


Figure 1. FPCs Channels

The key for this feature is the implementation of a new kind of Channel enabling the communication with Web Service. The real challenge is to create Channel as dynamic as possible, so it will be independent by content-type used by the HTTP call for transferring data (input or output, better *request* or *response*).

1. HTTP protocol and REST architecture

For understanding what is HTTP protocol it is convenient to start from the definition by *The Internet Society* document:

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [1].

This protocol enables so many functionality that now each device (mobile, personal computer but also appliances) can communicate using it.
Its standardization has allowed to develop a new protocol like SOAP (Simple Object Access Protocol) or REST (REpresentational State Transfer).
These have been adopted as architecture for exposing interfaces by server as services.

In the next part we analyze the messages used by HTTP protocol.

1.1 HTTP message structure

HTTP messages are divided into two type: request message and response message.
The request message is composed by three parts:

- request line – it begins with a method token, followed by the Request-URI and the protocol version [1].
- header – it includes some extra information (info about client, time of request, info about body)
- body – it is the content of request

Just below an example of HTTP request message:

```
POST /cgi-bin/process.cgi HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT)
Host: www.tutorialspoint.com
Content-Type: text/xml; charset=utf-8
Content-Length: 60
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

first=Zara&last=Ali
```

Notice, the body is not always required because Request-URI already specifics something like content. For example if you send a request to URL http://mysite.com/pages/contact, `/pages/contact` could be considered a param of request. This will be more clear in the part about REST architecture.

The response message is composed by three parts:

- status line – it consists of the protocol version followed by a numeric status code and its associated textual phrase [1]
- header – it includes some extra information (info about server, time of response, info about the body
- body – it is the content of the response, for example an HTML page or, more interesting for our case, a JSON message containing the temperature and the pressure of city specified in the request.

Just below an example of HTTP response message:

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed

<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

HTTP protocol has a powerful expression, but for standardizing its utilization like an interface has been introduced at the beginning SOAP and some years later, in the 2000, REST by Roy Fielding.
REST protocol is more defined and, especially in the last year, many web services born with RESTFull APIs (ex. Facebook, Twitter).

It is necessary to clarify a concept, Fielding thought a REST architecture applicable to any communication protocol not only for HTTP, but in these years it has become the natural low level protocol for implementing the REST architecture.
The next section explains how it is developed using HTTP protocol.

1.2 REST architecture and RESTFull APIs

The base principles of REST architecture are exposes in Fielding thesis [2] in Chapter 5:

Client-Server A uniform interface separates clients from servers.

Stateless No client context has to be stored on the server between requests.

Cache Clients can cache responses.

Uniform interface Simplify decoupling the architecture.

Layered System A client does not know the end-server that received request

Code-On-Demand Servers extend the functionality of a client by transferring executable code.

1.2.1 RESTFull APIs

Each Web Service usually specified a base URI, for example *http://myservice.com/api* and Internet media type (often JSON, but it is common XML too).
APIs are called with standard HTTP method (`GET`, `POST`, `PUT`, `DELETE`).
A client can interact with resource on Web Service using its URI (ex. *http://myservice.com/api/resources/id-10*) or with resources collection (ex. *http://myservice.com/api/resources/*).
The HTTP method specified the result expected:

GET Get the resource, so the body of HTTP response contains the model of resource. No content-type is required in HTTP request because all necessary information are retrieved by URI.
Multiple GET operations do not change server state.

POST Usually create and add a resource to collection. So body of HTTP request is required and probably also the content-type. HTTP response contains the resource aligned to server state (ex. server add an identifier to resource).
Multiple POST operations change server state (two times, two new resources).

PUT Put the resource and replace it. Body in HTTP request is required, but not in HTTP response. Client already known the resource, it needs just if operation ends correctly watching Status Code.

DELETE Delete the resource. So it is necessary just the URI of the resource nor body in the request, nor in the response.

The PerLa HTTP Channel has been implemented following these principles. So when in XML descriptor will be specified the requests and the responses for an HTTP channel, these will have to be consistent with RESTFull specification.
The next paragraph shows the generic Channel implementation in PerLa. Once known it, the implementation of HTTP Channel in PerLa will be easy to understand.

2. PerLa Channel design

In PerLa system the *virtual channel*, or simply channel, is that part enabling the communication between FPC and real device. In this context the device is represented by a common Web Service with a certain HTTP interface (ex. SOAP or JSON/REST).
The configuration of a channel is done out of PerLa implementation using an XML descriptor. So its instantiation has to be as dynamic as possible.
For doing it the PerLa design offers three base classes:

`AbstractChannel` is the abstraction of any communication channel between PerLa (FPC in particular) and the real device. We can consider TCP Socket technology, WebSocket, for advanced device, or HTTP protocol like a channel.
This object takes care of sendind a generic `Request`, creating a `Response` and adding it to queue passed by FPC.

`ChannelRequestBuilder` is object used by FPC for creating the requests to pass to channel. Each request class has its `ChannelRequestBuilder`.

`Request` is the object created at run-time when FPC needs to retrieve some data from real device. Usually, after to be created by `ChannelRequestBuilder`, it must be personalized with some parameters. For example a JSON/REST service may need to received the name of the city for retrieving a particular temperature.

The complete process of a performed call through the channel is showed by activity diagram in Figure 2.

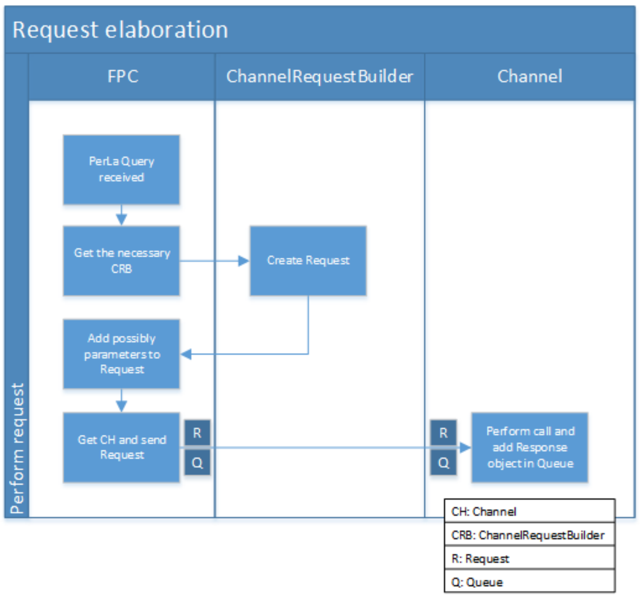


Figure 2. Request elaboration

Notice, usually the response inside the channel is synchronous and it is add to queue in common between channel and FPC. The FPC is responsible to decide if to use immediately the response (synchronous call) or to postpone this action and to retrieve it just when it is needed (asynchronous call).
The PerLa channel structure is build dynamically at runtime through the translation of a Descriptor (Java representation of XML descriptor) into PerLa objects.
This translation is done by a Factory. So exists:

`ChannelFactory` is responsible of XML channel descriptor validation. So it is used by FPC for creating Channel at PerLa start-up.

`ChannelRequestBuilderFactory` is responsible of XML request descriptor validation. So it is used by FPC for creating a `ChannelRequestBuilder` at PerLa start-up.

`ChannelDescriptor` is the Java representation of XML tag `channel`

`RequestDescriptor` is the Java representation of XML tag `request`

Once the PerLa Channel design is clear, let's to know how this design is used to implement HTTP Channel.

3. HTTP Channel implementation

HTTP Channel implementation has been realized using classes described before. So we will analyse step by step all classes.
Notice, all XML tags of HTTP context are refereed to *namespace* `http://perla.dei.org/channel/http`.

`HttpChannel` is that object responsible of performing HTTP call, using `HttpChannelRequest` like input and returning a `ChannelResponse`, standard PerLa object containing a payload.

`HttpChannelRequestBuilder` is that object responsible of `HttpChannelRequest` generation.

`HttpChannelRequest` is that object containing the real HTTP request with completed URL (host, path and query), HTTP method and eventually entity and content-type.
It will be change setting path, query and entity parameters.

`HttpChannelFactory` validates the channel descriptor and create the `HttpChannel` object.

`HttpChannelRequestBuilderFactory` is that object that parses the XML file, wrapped in `HttpRequestDescriptor`, and creates an instance of `HttpChannelRequestBuilder` for corresponding FPC.

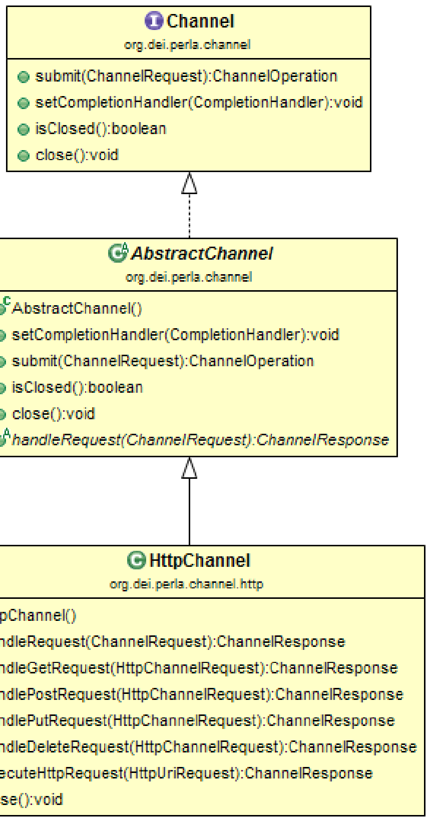
`HttpChannelDescriptor` is simple a channel instance responsible of the `HttpChannelRequests`.
It is a java object that represent `channel` tag having like attributes just an `id`.

`HttpRequestDescriptor` is Java class mapping HTTP `request` XML tag inserted in the device descriptor.

3.1 HttpChannel

This object is made by `HttpChannelRequestBuilderFactory` at start up of PerLa and associated to correspondent FPC. It is invoked by FPC using method `submit` that, consumed `HttpChannelRequest`, returns a `ChannelOperation` containing the logic with call result. This method is inherited by `AbstractChannel`, implementing `Channel` interface. The submit operation is done using `handleRequest` method, the core of `HttpChannel`.
The method dispatches the GET, POST, PUT and DELETE `HttpChannelRequest`, respectively, to `handleGetRequest`, `handlePostRequest`, `handlePutRequest`, `handleDeleteRequest`. Any method is implemented following REST architecture.
For a simple and standard implementation it has been used Apache HTTP Component library[4]. `CloseableHttpClient`, created into `HttpChannel` constructor, works as transporter of `HttpUriRequest`, built by PerLa Channel in *handle* methods.

In the Figure 3 is visible the class diagram of class in object.


Channel
org.dei.perla.channel
submit(ChannelRequest):ChannelOperation
setCompletionHandler(CompletionHandler):void
isClosed():boolean
close():void
AbstractChannel
org.dei.perla.channel
AbstractChannel()
setCompletionHandler(CompletionHandler):void
submit(ChannelRequest):ChannelOperation
isClosed():boolean
close():void
handleRequest(ChannelRequest):ChannelResponse
HttpChannel
org.dei.perla.channel.http
HttpChannel()
handleRequest(ChannelRequest):ChannelResponse
handleGetRequest(HttpChannelRequest):ChannelResponse
handlePostRequest(HttpChannelRequest):ChannelResponse
handlePutRequest(HttpChannelRequest):ChannelResponse
handleDeleteRequest(HttpChannelRequest):ChannelResponse
executeHttpRequest(HttpUriRequest):ChannelResponse
close():void
Figure 3. Class diagram of HttpChannel

3.2 HttpChannelRequestBuilder

The `HttpChannelRequestBuilder` object is used by FPC to create an `HttpChannelRequest` before invoking an HTTP call. It is built at start up and gets all needed parameters to set when an `HttpChannelRequest` will be created and the id of Channel to use for sending this request.
Made by `HttpChannelRequestBuilderFactory`, it instances an `HttpChannelRequest` through `create` method.
In the Figure 4 there is its class diagram.

3.3 HttpChannelRequest

`HttpChannelRequest` models the content sending through the channel, so has essentially four parameters:

- HTTP method
- Uri
- Content-type
- Entity

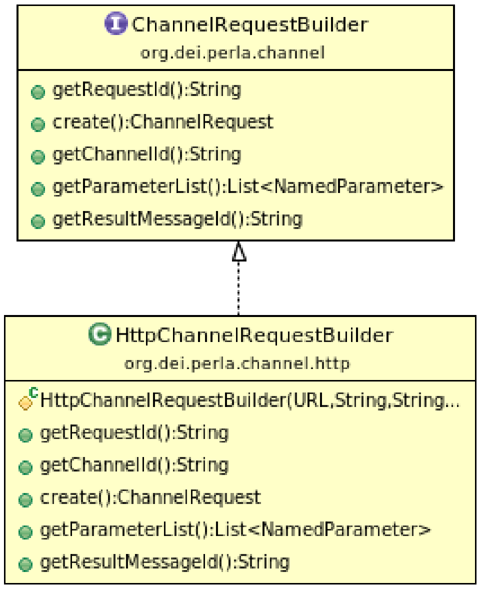

ChannelRequestBuilder
org.dei.perla.channel
getRequestId():String
create():ChannelRequest
getChannelId():String
getParameterList():List<NamedParameter>
getResultMessageId():String
HttpChannelRequestBuilder
org.dei.perla.channel.http
HttpChannelRequestBuilder(URL,String,String...
getRequestId():String
getChannelId():String
create():ChannelRequest
getParameterList():List<NamedParameter>
getResultMessageId():String

Figure 4. Class diagram of HttpChannelRequestBuilder

Except HTTP method, a part of Uri and Content-type, defined in the XML descriptor, the others are setted dynamically by FPC using `setPayload` method. It uses a string as identifier, for query url, additional uri path and entity, and `ChannelPayload` object, containing the value.
Query url and uri path are encapsulated in Uri parameter so `HttpChannel` can be used it already formatted.

`HttpChannelRequest` implement methods of `ChannelRequest` interface used by FPC on a level abstracter (cfr. Figure 5).

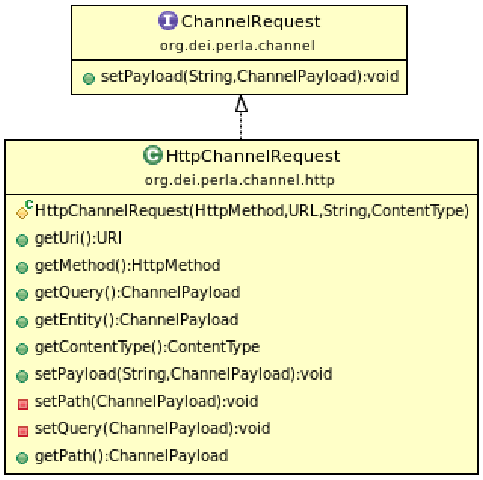

ChannelRequest
org.dei.perla.channel
setPayload(String,ChannelPayload):void
HttpChannelRequest
org.dei.perla.channel.http
HttpChannelRequest(HttpMethod,URL,String,ContentType)
getUri():URI
getMethod():HttpMethod
getQuery():ChannelPayload
getEntity():ChannelPayload
getContentType():ContentType
setPayload(String,ChannelPayload):void
setPath(ChannelPayload):void
setQuery(ChannelPayload):void
getPath():ChannelPayload

Figure 5. Class diagram of HttpChannelRequest

3.4 HttpChannelFactory

This class is responsible of `HttpChannel` creation from `HttpChannelDescriptor`. It validates the descriptor and initializes the channel applying the attributes (ex. `id`, channel identifier). At current art state it sets just id of the channel because HTTP connection has no so many parameter to specify for working.
In the future is possible to introduce some attributes, for example, to change the TCP/IP default port (80 for HTTP protocol) or something else.

method	response	content-type	entity
GET	M	NA	NA
POST	M	M	M
PUT	NA	M	M
DELETE	NA	NA	NA

Table 1. HttpRequest Attributes consistency

3.5 HttpChannelRequestBuilderFactory

It is one of the critical point of HTTP integration because `HttpChannelRequestBuilderFactory` must validates the XML `request` tag an checks its consistency with REST architecture. In Table 1 are shown which attributes are allowed (A), not allowed (NA) and mandatory (A) for each HTTP Method.
Notice, attributes `id`, `channel-id` and `host` are always mandatory while attributes `path` and `query` are always allowed.

All controls are done into `create` method that throw an `InvalidDeviceDescriptorException` exception when some control is not passed.

3.6 HttpChannelDescriptor

`HttpChannelDescriptor` is that POJO translating the XML `channel` tag. At the moment represents just one attribute, the `id` of the channel. The class of channel is already specified by name-space into XML.
Notice, channel identifier is used for associating each request with is channel. Just below there is an example of `channel` tag:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<device ...
  xmlns:ht="http://perla.dei.org/channel/http">
  ...
  <channels>
    <ht:channel id="http_ch_01">
    </ht:channel>
  </channels>
</device>
```

3.7 HttpRequestDescriptor

Request descriptor is a POJO that maps all tags of `request` tag. For understanding the mapping is important to know how XML `request` tag is composed. All `request` tags are characterized by nine attributes:

`id` is a string identifier of request.
Required

`channel-id` is a string identifier of channel sending this request.
Required

`host` is the host for sending HTTP request. It is accepted also a complex url like

`http://mysite.com/one/path?q=i`
Required

`path` is the identifier of tag `message` that represent a dynamic path.
Optional

`query` is the identifier of tag `message` that represent a dynamic query.
Optional

`entity` is the identifier of tag `message` that represent the entity (content) of HTTP request
Required for POST and PUT request

`method` is an enumeration value that specifics HTTP method for the request (get, post, put or delete).
Optional, default is get

`content-type` is the content-type specified in HTTP request.
Optional, default is `*/*` (Known as *wildcard* content-type)

`response` is the identifier of tag `message` that represent the response content of HTTP request
Required for post and get request
This is a POST request example:

```
<?xml version="1.0" encoding="UTF-8"?>
<device ...
  xmlns:ht="http://perla.dei.org/channel/http">
  ...
  <requests>
    <ht:request id="post_req"
      channel-id="http_ch_01"
      method="post"
      host="http://mysite.com"
      path="req_path_id"
      query="req_query_id"
      response="req_response_id"
      content-type="application/x-www-form-urlencoded"
      entity="req_entity_id"/>
  </requests>
</device>
```

Now it is simply to watch class diagram in Figure 6 and recognize the relationship between methods and attribute of `request` tag.
`HttpRequestDescriptor` implements the `getMessageIdList`, it is an abstract one inherited by `RequestDescriptor`, usable by FPC to parse the request.

4. Conclusion

This implementation allows an integration of PerLa with SOAP and JSON/REST services. These technologies could utilized the same Channel despite the REST orientation of HTTP Channel code. It is possible because SOAP calls use, essentially, just GET and POST HTTP method.
The next challenge is to define the translation between FpcMessage (referenced by `path`, `query`, `entity` and `response` attributes).

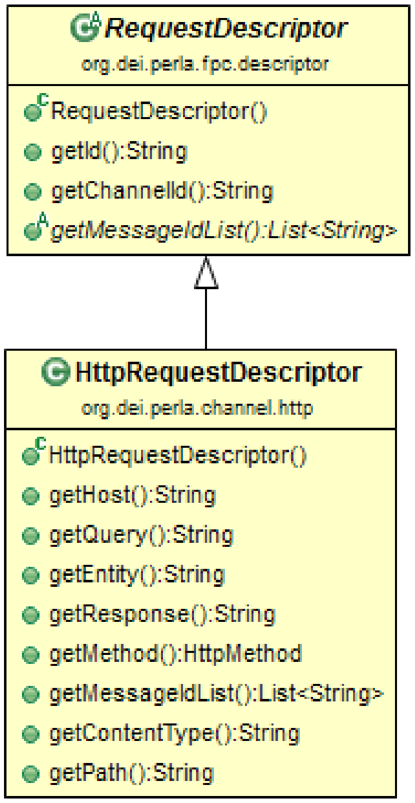


Figure 6. Class diagram of HttpRequestDescriptor

A possible scenery is described into Introduction section. It showed how an FPC could be extended with virtual sensors given by Web Service.

This is the first simple idea, but in these years, with the birth of open-data the possible scenarios become more complex.
Thinking about public transport in a city, PerLa takes information about the bus time-table, through a Web Service, and can:

- notify if the next is the last ride
- suggest to use bus (maybe with a touristic ride) or underground depending on weather

In summarize to open PerLa to Web Service world enables the integration between the small data of sensor and the big data of web.

References

[1] R. Fielding J. Gettys J. Mogul H. Frystyk L. Masinter P. Leach T. Berners-Lee *Hypertext Transfer Protocol – HTTP/1.1* 1997: The Internet Society. *http://www.w3.org/Protocols/rfc2616/rfc2616.html*

[2] R. Fielding *Architectural Styles and the Design of Network-based Software Architectures* 2000: University of California, Irvine. *http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm*

[3] F. A. Schreiber, R. Camplani, M. Fortunato, M. Marelli, G. Rota *Design of PerLa, a Declarative Language and a Middleware Architecture for Pervasive Systems* 2010: Politecnico di Milano.

[4] *Apache HTTP Components* The Apache Software Foundation.
https://hc.apache.org/