HttpDispatchThreadingDesign

Threads and Synchronization in HttpDispatch

About

The purpose of this document is to provide a design documentation for the use of threads and synchronization in HttpDispatch that is separate from the source code. Unlike the source code, this design document will not only reflect the current implementation, but also lists design alternatives and gives a rationale for design decisions. And there are pictures here!

Note that HttpDispatch is the working title for what was formerly referred to as HttpAsync.
There are some leftover references to the old name on this page, in particular the labels in the pictures.

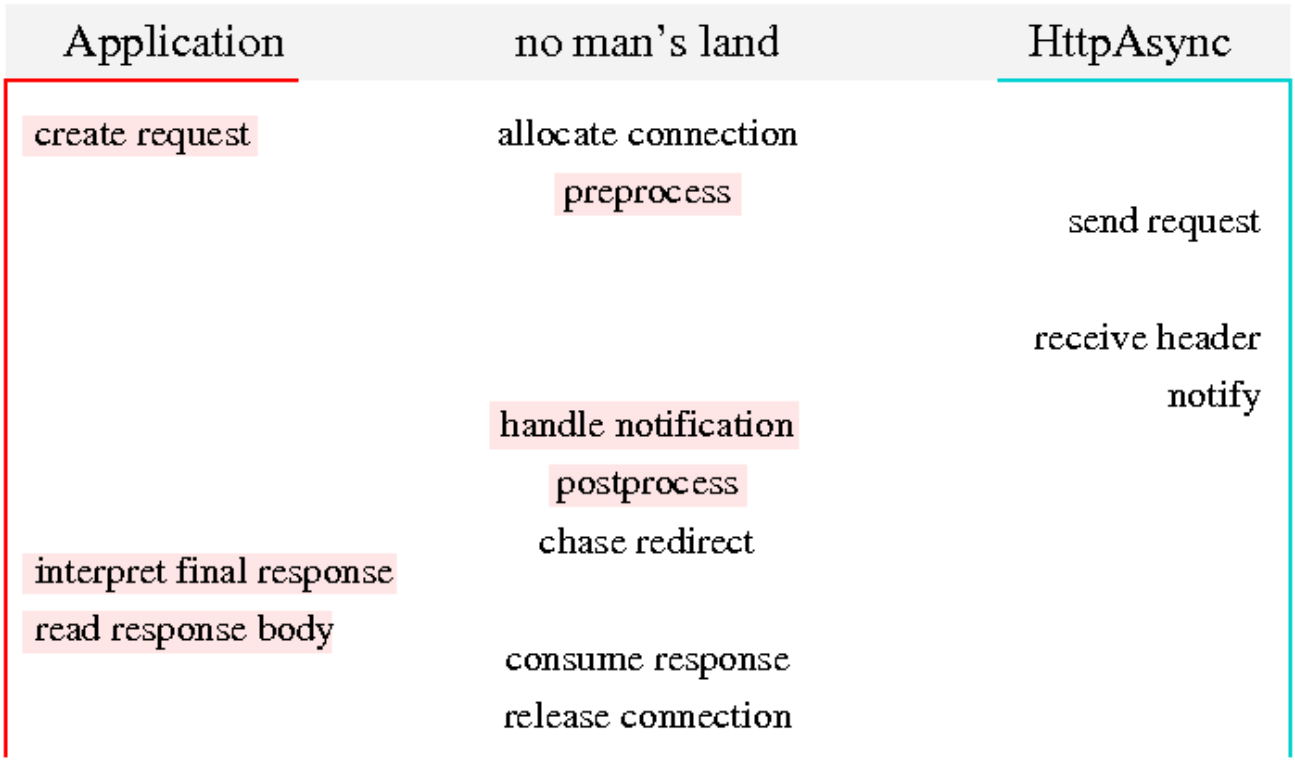
Work on HttpDispatch is currently suspended. The code mentioned below is archived here. It compiles against HttpCore alpha 4. A lot of progress has been made in HttpCore and HttpConn since it was originally developed. The code is therefore outdated, but can still serve as a starting point to pick up development. If you feel like spending time on HttpDispatch, just send a mail to the developer list.

Background

The purpose of the HttpDispatch component or module is to provide an API that allows applications to execute HTTP requests asynchronously. That means the application creates a request, hands the request over to HttpDispatch, and later picks up the response. Typically, applications also want to be notified when a response becomes available. There is a selection of UseCases that address asynchronous communication.

There will always be at least two threads required, one on the application side and one background thread on the HttpDispatch side. On this high level of abstraction, it doesn't matter whether there are one or many threads on either side. There may also be several applications using HttpDispatch at the same time, or several components of one large application.

Executing a request involves several steps. Each step needs to be executed by either an application thread or a background thread (from HttpDispatch). As part of the design, it is necessary to define which step should be executed by which kind of thread. Although it is possible to defer such decision to runtime, threading issues will be easier to handle if the assignment is static. The following figure shows the steps required to execute a request.



Steps that necessarily have to be executed by an application thread are shown to the left. Only the application can decide which request should be executed and what to do with the response.
To the right are steps that have to be executed by a background thread. Sending of the request and waiting for the response is there since it is the purpose of HttpDispatch to offload such tasks from applications. Notification for incoming responses has to be triggered by the thread that was waiting for the response. Receiving the response header is assigned to the background thread too, because it is a precondition for notification, as explained below. The steps in the middle column can reasonably be assigned to either side.

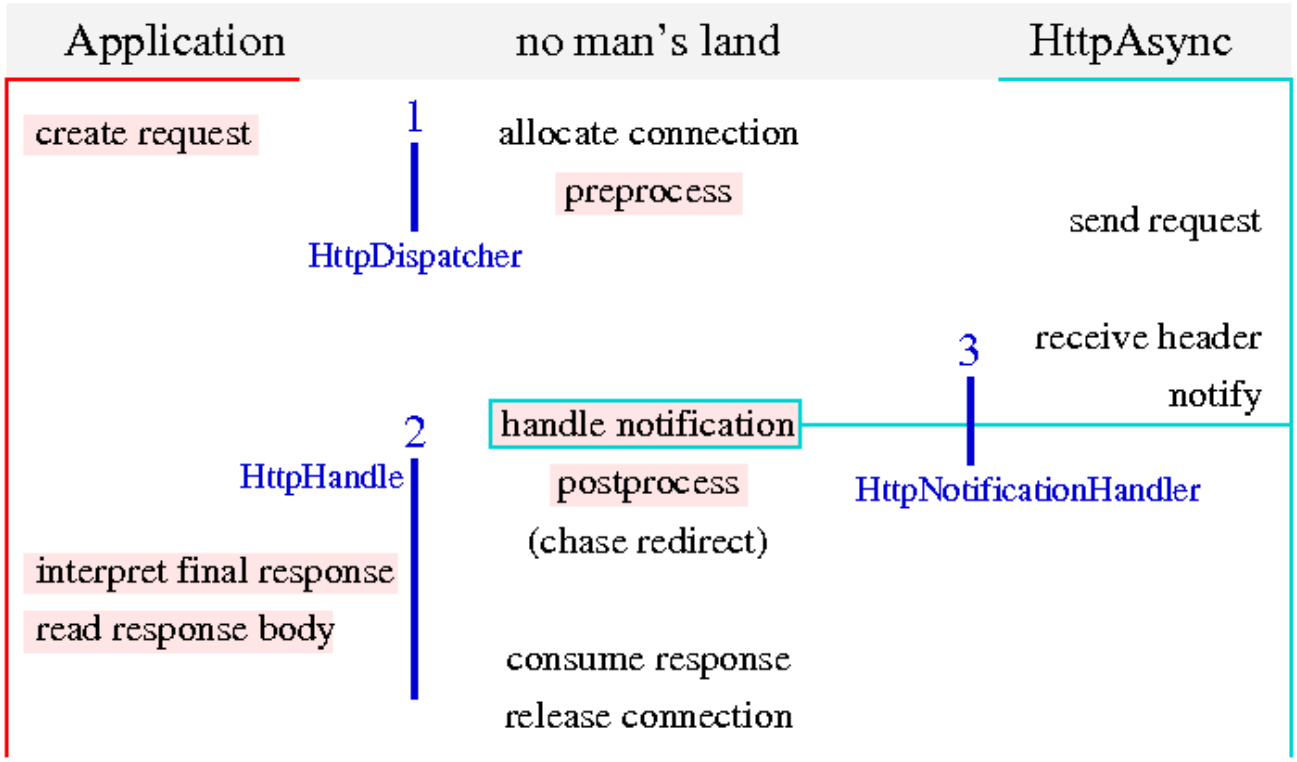
Assigning the steps to application threads or background threads is one thing. Another question is the responsibility for the code that gets executed. Some of the steps in no man's land are implemented by application code, indicated by the red backdrop. While the code for the pre- and postprocessing is not necessarily written by the application developer, it is the application that decides which interceptors will be executed in these steps. Interceptors are also a plugin point for application code, therefore the responsibility for what is done in these two steps is with the application. It is arguable whether "send request" should be considered application code, since it can involve a request entity provided by the application developer. In HttpClient, the request entities included with the package were usually sufficient, so this step is not marked as executing application code here.

The order of the steps from top to bottom is roughly chronological, but some are independent and can be executed in a different order. For example, a request must be created before it can be preprocessed. But the connection for sending the request can be allocated before or after preprocessing, or even before the request is created. The table below shows the sequences in which some of the steps have to be executed, one sequence in each column. Postprocessing has to be done before chasing redirects, since there might be cookies in the response that need to be stored for the followup request. Reading the response header should be done before notification, because a notification before status code and headers of the response are known would be very inconvenient to use. The other sequences are obvious.

create request preprocess send request receive response header postprocess interpret final response
allocate connection send request receive response header read response body consume response release connection
receive response header notify handle notification
receive response header postprocess chase redirects

API

The application programming interface (API) for HttpDispatch in package `org.apache.http.async`
defines three interfaces. The following figure shows their place with respect to the steps that have to be executed.



Two of the interfaces are application-facing. `HttpDispatcher` is used to transfer control over a request to HttpDispatch. Since this is done by a call from an application thread, the implementation can then execute code in that application thread. Eventually, the request has to be passed to the background threads that handle the asynchronous communication. The application obtains an instance of the second interface as a result of the call to `HttpDispatcher`.

Instances of `HttpHandle` are specific to a request. When the application tries to access the response to a specific request, it does so through the `HttpHandle` for that request. When the application is done with processing a response to a specific request, it indicates that to the `HttpHandle` for that request. If the application has to cancel a specific request, it does so through the `HttpHandle` for that request. Again, the implementation has the opportunity to execute some of the steps in the calling application thread. Thread synchronization is a particular issue here, since several application threads may be calling the same instance of `HttpHandle` concurrently.

The third interface `HttpNotificationHandler` is used by background threads to notify applications of incoming responses, or of problems encountered while executing a request. It would have been possible to define notifications in terms of specific objects for thread synchronization. While background threads would not have had to execute application code for notification in that case, the flexibility for application developers would have been signifcantly reduced. Instead, a background thread is calling directly into application code, which can then use suitable means to relay the notification to application threads. The thread calling into application code is symbolized by the cyan border around the red box for "handle notification". Implementing the `HttpNotificationHandler` interface requires **special care** by application developers, since a misbehaving notification handler can take down background threads and thereby stall other requests as well.

The step "chase redirect" is shown in brackets since it is not yet part of the API. If it becomes part of the API, it will probably not be in the `HttpHandle` interface, although it's position in the figure might trick you into expecting that. There are too many problems to be solved first, so let's not worry about chasing redirects now.

Synchronization Details

`HttpDispatcher` has a method `sendRequest` to transfer control of a request and obtain a handle. `abortAll` can be used to cancel all request (handles) currently controlled by the dispatcher, but it leaves the dispatcher operational.
`shutdown` (*not yet implemented*) will cancel all requests and stop operation of the dispatcher. It releases resources such as background threads. Dispatcher implementations may have methods that allow reinitialization, but that is not part of the interface.

`HttpHandle` has a method `awaitResponse` which will block the calling process until the response is available or until an error is encountered. By using notifications, the caller can make sure that it will be blocked only momentarily, if at all.

`close` indicates that processing of the response has finished and that the connection over which the response is being received can be used for another request. When the handle is closed while the response has not been read completely, the rest of the response may be consumed.

`abort` can be called at any time to abort processing of the request. If the request is not yet sent, it will be removed from the relevant queue gracefully. If it is sent but the response not yet received, the response will be discarded. Aborting a handle never consumes the rest of the response, but it has a negative effect on keep-alive and pipelining. After being aborted, the handle behaves as if an error was encountered.

`isLinked` indicates whether the handle is still linked to the dispatcher and it's connection. Closing or aborting the handle will unlink it. Note that access to `isLinked` can not be synchronized:
even if it returns true, you can't be sure that the handle is still linked by the time you call another method. Once a handle is unlinked, it remains unlinked.

`HttpNotificationHandler` has methods `notifyResponse` and `notifyProblem`, which are called for incoming responses and encountered problems, respectively. There will be at most one notification for either the response or a fatal problem. If `notifyResponse` is called but throws a runtime exception, that is a fatal problem. But there will be no problem notification, since the response notification has already been given. On the application side, the handle will behave as if an error was encountered.

There can be several notifications about non-fatal problems before the final notification, but not afterwards. Imagine a server that receives the request header, sends an error response immediately, and closes the connection while the dispatcher still tries to send the request body. This triggers an exception on sending, but the response from the server is available. `notifyProblem` may be called for a non-fatal problem then. It's return value indicates whether the problem should be handled as a fatal one, or whether processing should resume and another notification given.

Notifications are triggered exclusively by operations of the background threads. Aborting a request at any time does *not* trigger a notification, even though the handle will behave as if an error was encountered.

All methods in `HttpDispatcher` and `HttpHandle` are thread safe. All methods in `HttpNotificationHandler` must be thread safe. They also must return quickly to keep the background threads available for tasks related to other requests. In particular, none of the blocking or time-consuming methods of `HttpHandle` must be called during a notification.
`HttpHandle.abort` is OK to be called. Some implementations may also allow
`HttpHandle.close` to be called, but that is not guaranteed by the API.

Application Considerations

Applications using HttpDispatch have one very important responsibility which was not been mentioned so far. It may sound trivial, but really it isn't:

Applications **must** process responses as they arrive.

Due to the asynchronous nature of HttpDispatch, an application can generate several requests and pass them to a dispatcher. HttpDispatch does *not* guarantee that these requests will be sent in order. Responses may arrive in any order (even different from the order in which requests are sent), and each response with an entity locks up one connection until it is processed.

Theoretically, notification is optional. An application thread can block on the handle for a request until that specific response arrives. But since the order in which requests are sent is not guaranteed, it can happen that other responses which are not processed by the application lock up all connections, and that the one request on which the application waits will never be sent. Even if this deadlock scenario does not occur, blocked connections will degrade performance.

Probability theory tells us that what can happen will happen eventually. Murphy's Law tells us that what can go wrong will go wrong, in the worst possible moment. Therefore, applications that generate more than one request per thread at a time **must** use notification in order to process responses on arrival.

Blocking IO Implementation

This section presents design alternatives for implementing the HttpDispatch interfaces. An implementation is also referred to as a *dispatcher*, since each implementation of `HttpDispatcher` requires a matching implementation of `HttpHandler` and will make use of `HttpNotificationHandler`, which is implemented by applications.

In the figures below, fat lines indicate threads running from top to bottom. This is not necessarily one thread on either side. The fat red line to the left stands for all application threads, while the fat cyan line to the right stands for all background threads. Objects for thread synchronization are represented by a queue-like symbol. Thinner lines in the respective color connect the synchronization objects to the thread lines. Big queue objects are used for passing handles, small queue objects for synchronizing on a specific handle.

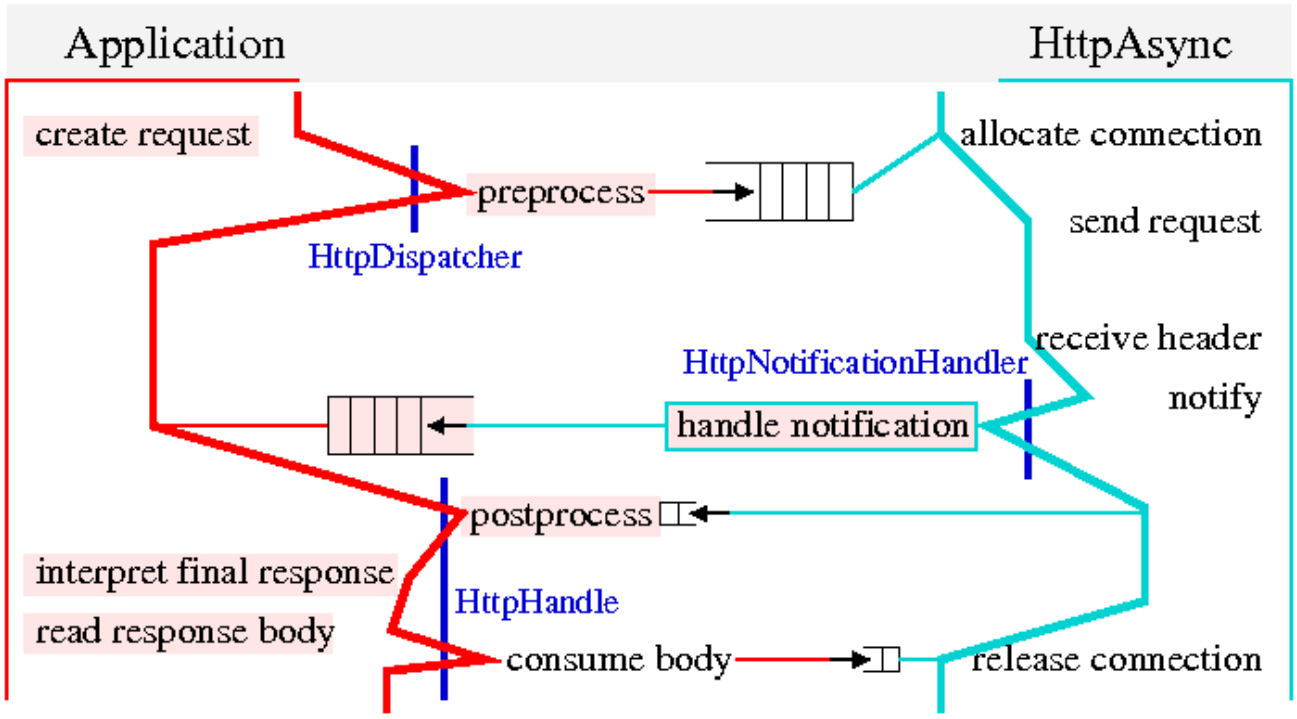
There are two big queue symbols in each design alternative. One is used to pass the handles for newly created objects from the application side to the background threads. That object is under control of the dispatcher. The second one is used to pass handles from the notification handler to the application side. That happens under control of the application, indicated by the red backdrop of the symbol. Applications can use any number of actual objects there, for example to route handles to different application threads.

There are two small queue symbols in each design alternative. One is used to pass the response (or error) from the background threads to the application threads. The other is used to indicate completion of response processing to the background threads, which can then release or re-use the connection that was locked up by that response. Both of these synchronization objects are under control of the dispatcher.

Red Design

This extreme design is based on the following premises:

- Background threads are a shared resource that should be used only for what is absolutely necessary.
- Application code is unstable and should be executed by application threads whenever possible.



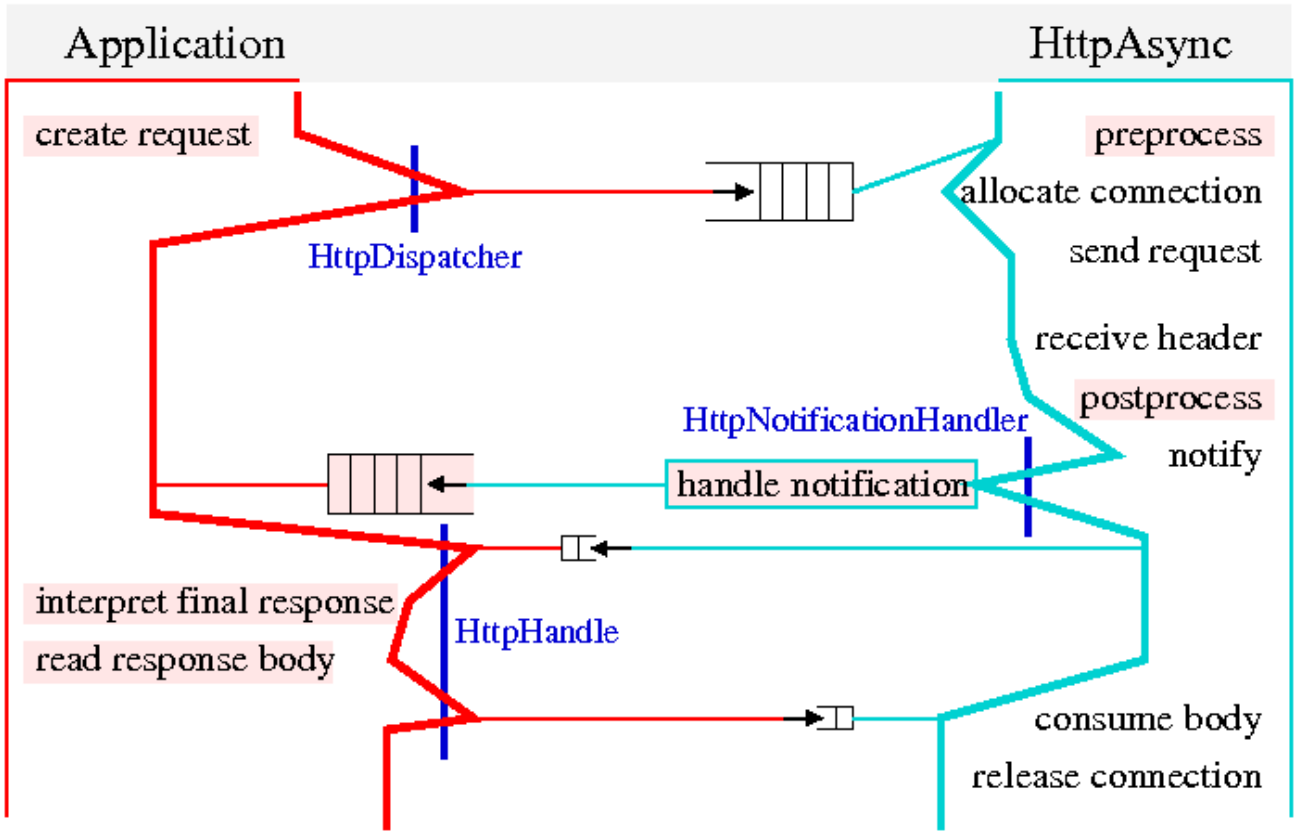
Preprocessing and postprocessing is done by application threads because these steps execute application code. Consuming the response is also done by an application thread, because it is a potentially long-running task that does not necessarily have to be executed by a background thread.

With this design, notification handling does not have access to the postprocessed response. The notification handler can not close the handle either. Errors in preprocessing will not generate load in the background threads. The code for pre- and postprocessing can use blocking operations, including user interaction. Only an application thread will be blocked, but the dispatcher continues operation.

Cyan Design

This extreme design is based on the following premise:

- If it can be done by a background thread, let it be done by a background thread.



Preprocessing and postprocessing are done by background threads, as is consuming the response. Postprocessing is done before notification, since that is the last chance to detect and report a problem in a background thread.

The notification handler has access to the postprocessed response, and it can close the handle. Errors in preprocessing will trigger a problem notification. Pre- and postprocessing are subject to the same restrictions as notification handling. In particular, they can not use long-running blocking operations, since they would block a background thread and thereby interfere with processing of other requests and responses.

Consolidated Design

After discussion on the developer mailing list, the following design choices have been made for the initial implementation. They are subject to review, discussion, and change.

1. Preprocessing can be switched between application thread and background thread through a parameter.
   The default is to preprocess in the application thread, since that keeps bad requests that fail to preprocess out of the dispatcher.
2. Postprocessing can be switched between application thread and background thread through a parameter.
   The default is to postprocess in the background thread, since it is unpredictable which of several application threads would be the one that does the postprocessing.
3. Consuming of the remaining response body is done in the background thread, since that step is logically tied to connection management.
   Applications that don't want the background thread to consume the response body can consume it explicitly before closing the handle.

Non-blocking IO Implementation

The blocking IO implementation promises maximum performance. It's major drawback is that it requires at least as many background threads as there are connections, since a dedicated thread needs to wait for incoming responses on each connection. That may be acceptable in client applications, for example a web spider. For server side applications like proxies, this resource inefficiency is typically not acceptable.

Non-blocking IO allows a single thread to wait for an incoming message on *any* connection. Although it is possible to switch sockets between blocking and non-blocking modes, this can not be used to mix non-blocking IO for waiting with blocking IO for receiving. The socket behavior can only be specified for both directions, sending and receiving. When pipelining, the socket can be used for sending requests at any time, the operation mode must therefore not be changed. An extra mixed-mode dispatcher that excludes pipelining hardly seems worth the effort.

This is the place for discussing `java.nio` based dispatchers.

The foundation for implementing HTTP communication with NIO is in HttpNIO.