ClientConnectionManagementDesign

This page is obsolete. The redesign of the connection management code in HttpClient has been completed

Introduction

Purpose of Connection Management

1. Enforce connection limits.
   Typical limits are a maximum number of connections in total, and a maximum number of connections to a target host.
2. Re-use open connections to increase performance.
   Opening a connection is time consuming, in particular for TLS/SSL connections. The connection keep-alive feature of HTTP allows for multiple requests to be sent over a connection.

Terminology

Although the definitions talk about *sending*, communication is always bidirectional. After an HTTP request is sent, the response needs to be received on the same connection.

target

${renderedContent}

proxy

${renderedContent}

connection

${renderedContent}

tunnel

${renderedContent}

route

${renderedContent}

Analysis

Usage Patterns

In the traditional usage pattern, a thread that needs to send an HTTP request to a target will ask the connection manager for a connection to that target. The thread blocks until a connection becomes available. If the connection is already openend to the target, it is used immediately. Otherwise it needs to be opened first, which might require tunnelling a TLS/SSL connection through a proxy. When the thread is done with the HTTP communication, the connection is returned to the connection manager. If there are no side conditions that prevent re-use, the connection is kept open for a limited time so that subsequent requests to the same target can be sent immediately.

HttpCore-NIO: The NIO module of HttpCore hides the details of NIO communication from applications. The usage pattern will be similar to the traditional pattern, except that the application thread will not establish a connection itself. IO is performed by a background thread.

HttpDispatch: In HttpDispatch, the traditional usage pattern would be reversed. Instead of requests looking for a connection, there are connections looking for requests. Requests generated by the application will be collected in a pool, the application threads don't block. Background threads will choose requests that can be sent over an open connection. If there are none for that target, the connection can be closed and re-openend to a different target in order to send other requests.

Connection Re-use

Connection re-use is subject to a keep-alive strategy. Both client and server can decide that a connection should not be kept alive. However, a connection can only be re-used if some preconditions are met, which are discussed in the following sections. If the keep-alive strategy can determine from the request alone that the connection can not be re-used, the preconditions don't matter. If the preconditions are not met, the keep-alive strategy based on the response doesn't matter.

Routing State

Opening a connection for a route can require several steps. The type of route that is established determines for which routes the connection can be re-used.

closed

${renderedContent}

direct to target

${renderedContent}

plain to proxy

${renderedContent}

tunnelled via proxy

${renderedContent}

Reusability can be affected by the authentication state of a connection. If TLS/SSL with client authentication is used, a connection identifies the user to the target. With NTLM authentication, the connection identifies the user to the proxy or target. Re-use of an authenticated connection for requests from a different user is a security breach. While TLS/SSL client authentication is out of scope for HttpComponents, NTLM authentication is not and needs to be considered.

Reusability can also be affected by the local host address being used. On machines with more than one network interface, applications might have specific requirements as to which of the interfaces is used for communication.

Communication State

See this discussion in the mail archive.

Implementation Review (HttpClient 3.1)

The `SimpleHttpConnectionManager` in HttpClient 3.x has exactly one connection. It is re-used only for exactly the same route. No limits need to be enforced.

The `MultiThreadedHttpConnectionManager` in HttpClient 3.x keeps a pool of connections. Re-use depends on exact matches of the route. Limits are enforced per route. In other words, the maximum connections per host are interpreted as maximum connections per route. If multiple proxy servers are available, the maximum per host can be exceeded by using different routes to the target host. This has not been a problem in the past.

Once a connection is obtained from the manager, it can be used for any target. As long as it is allocated it will count as a connection along the route for which it was obtained. The `HttpMethodDirector` takes care to use connections only for the route for which they are obtained. When following redirects on a different route, the connection is released and a new one gets allocated.

A more sophisticated connection manager could provide performance benefits for HttpClient 4.x. In particular, re-use of plain connections to a proxy for all routes via that proxy has potential. Even more so if the proxy requires NTLM authentication and the previous authentication state can be re-used.

Adherence to connections per host limits on a per-host basis requires changes to the connection management logic. Enforcement of connection usage exclusively along the advertised route is only required when dealing with misbehaving applications.

Design Ideas

Connection Management

Connections should be requested from a connection manager with a specification of the intended route. The connection manager returns a connection which is either closed, matches the route exactly, or is a prefix of the route. We need representations for the intended route and for the actual route of the returned connection. Connection managers can compute internal keys for connection lookup based on the routes.

A connection manager can not establish all routes automatically. Opening a plain socket or TLS/SSL socket could be implemented there, but tunnelling requires HTTP communication and possibly HTTP authentication. This is out of scope for a connection manager. Since routes can not always be established by the connection manager, that responsibility should reside elsewhere completely, rather than being split between the connection manager and an outside component. That does not mean a connection manager can't offer methods for example to open sockets, but the responsibility for calling these is elsewhere.

When a connection is released, the connection manager needs to know which route has actually been established. The connection may be closed, the full route may have been established, or a prefix thereof. Since routes are established outside of the connection manager, there might also be misbehaving applications that establish a route different from the one for which the connection was allocated.

Tracking of the steps can be done by the application, by the connection, or by the connection manager. Tracking by the application opens a gaping hole for connection management problems that are very hard to analyse and debug. Tracking by the connection or by the connection manager is functionally equivalent. The connection could call back to it's connection manager, and the connection manager would require the connection as an argument. This kind of tracking requires a set of methods that correspond to the steps in establishing a route:

- open connection to target/proxy
- tunnel connection to target
- layer (secure) connection over tunnel
- authenticate against target/proxy (connection based authentication only)

Optionally, establishing of a route different from the intended one could be prevented by the connection or connection manager.

Routes

The two most common type of routes have one or two hops, being either direct or through exactly one proxy. However, there are cases when connections have to be established through a chain of proxies (HTTPCLIENT-649). A modifiable route representation is useful for tracking. A non-modifiable route representation could be useful for lookup keys and return values. A sophisticated connection manager will probably need different lookup keys though. Options:

- both a modifiable and non-modifiable implementation similar to `StringBuilder` and `String`, optionally with a common interface
- modifiable route representation only, similar to `HostConfiguration` in HttpClient 3.x

The target and proxy can be represented by `HttpHost` objects. This identifies schemes (HTTP, HTTPS) by their name. It does not cover cases where applications use different socket factories for the same scheme. That's not likely to be a problem.

Route length (direct/proxied) for the two common cases, tunnelling state and layered/plain can be represented by `boolean` values. Using `int` for the route length would allow for custom implementations of more complex routes (with the interface option), but the restriction to two hops will probably show up elsewhere too. Multiple levels of layering are hypothetical and therefore not considered.

Generic `Object` instances can be used to represent connection based authentication state. Two objects are required for proxy and target authentication, assuming at most one proxy. Comparison relies on the generic `equals` method. Applications can use whatever is considered appropriate to represent the authentication, for example:

- `CredentialsProvider` for NTLM authentication. If a route is requested with the same `CredentialsProvider`, the same credentials would be available and so the authentication state can be re-used.
- `SSLSocketFactory` for TLS/SSL with client authentication. The socket factory uses a unique key store, so the same credentials would be available and the authentication state can be re-used.
- A key store for TLS/SSL with client authentication. This is a variation of the `SSLSocketFactory` option.
- application specific user object for applications dealing with more complex multi-user scenarios

The drawback of using `Object` to represent authentication state is that only a single level of authentication can be used. Multiple connection based authentication levels, for example adding NTLM authentication on top of TLS/SSL with client authentication, would not be covered. (No, that example doesn't make sense.)

Alternatively, an interface for the authentication state could be defined with two comparison methods for checking whether a state is reachable by means of *upgrading* from another one. Ease of use in the most common situations can be achieved by a default implementation that wraps an `Object` and maps both methods to `equals` on that object. Two methods are needed so that both authentication states, the current one and the intended one, can decide that the upgrade is not permissible.

Modifiable objects for authentication states have implications if used in a non-modifiable route representation. This would need to be documented carefully, but the problem is not different from using modifiable objects as lookup keys.

Connection Release

Connections need to be released to the manager. This should be done in a timely fashion as soon as they are no longer needed. A connection is no longer needed when the response entity (if any) will not be read from anymore. That can happen because it is read or buffered completely, because the application doesn't need to read the rest, or because of an error that prevents the rest of the entity from being read.

An application can release the connection explicitly, but that puts the burden of a timely release on the application. It would be preferable to auto-release connections in certain situations, for example if there is no entity, if the entity is read completely, or if the stream for reading the entity is closed. In error situations that are deemed non-recoverable, a stream could also auto-release the connection.

Tricky to detect are situations where applications just don't make use of the entity, without an API call to indicate that. If the reference to the entity is lost, garbage collection of the entity or connection can be detected. Cases where the application keeps a reference could be dealt with by a timeout mechanism, where the application is given a maximum handling time or idle time until the connection is auto-released.

See also this discussion in the mail archive.

Status

HttpClient 4.0 alpha3

Classes `HttpRoute` and `RouteTracker` are immutable and modifiable representations of a route. Route information includes flags whether the route is tunnelled, layered, or secure. Route information excludes connection based authentication state. The `ThreadSafeClientConnManager` (TSCCM) uses the same route-based management approach as the `MultiThreadedHttpConnectionManager` (MTHCM) in 3.x. An `HttpRoutePlanner` is used to determine the route needed to execute a request in advance. That route is then requested from the TSCCM. Connection release can be handled on the connection level, the entity level, and the stream level. There is a generic optional interface that is implemented on these levels. An auto-close input stream similar to the one in 3.x takes care of the default release, a managed entity can be used to close or abort the connection if the stream is not available.

This implementation seems to work well enough, with the same limitations as MTHCM in 3.x. The first obvious problem is that it still ignores connection based authentication state (CBAS). The idea is to add CBAS as an additional information that the TSCCM has to consider when selecting available connections from a pool. The impact of this change on the connection management code remains to be assessed. (HTTPCLIENT-652)

A second problem is the missing support for upgrading security over an existing plain HTTP connection, as specified by RFC 2817. Since the security flag is part of the route, the route of the connection after an upgrade is different from the route before. TSCCM is not equipped to handle this distinction. It cannot return an insecure route for upgrading when a secure route is requested. Support for upgrading security might be squeezed in by always requesting an insecure connection, and by managing the insecure and secure connections in the same pool of TSCCM. This is very similar to the CBAS problem, as the (in)security of the connection should be considered when selecting available connections from a pool. (HTTPCLIENT-750)

Also, the `HttpRoutePlanner` is expected to determine the route, including security, in advance. This is incompatible with the notion that security of a connection is a runtime property, which may depend on the target host (intranet/internet) or the strength of a cypher suite chosen for TLS/SSL.

In summary, the current implementation suffers from the same shortcomings as the 3.x one. If support for CBAS is added, we should reconsider the approach for route representation and maybe manage the flags (tunnelled, layered, secure) separately from the route and together with the CBAS.