# WebSocket Throttling Design

Adam Rice <ricea@chromium.org>

## Objective

Implement RFC6455-compliant throttling for the new WebSocket stack in net/.

WebSockets multiplexed over SPDY or HTTP/2 transports are out of scope for this document (except for WebSockets tunnelled over a SPDY proxy, which is just an ordinary proxy from the WebSocket point-of-view).

## Background

The new WebSocket stack re-uses the HTTP stack for its handshake. Unfortunately, the connection throttling semantics differ between HTTP and WebSockets:

- For HTTP, the browser limits itself to 6 connections per (hostname, port) pair.
- For WebSocket, only one connection a particular (ip, port) pair may be in the handshake phase at a time. This permits connection limits to be enforced on the server side, so there is no theoretical limit on the number of connections a browser may have to a particular server.

When connecting via a proxy, the browser doesn't usually have the IP address, so it is expected to serialise connections by (hostname, port) instead, and in addition keep some limit on the overall number of connections in the handshake stage to prevent DoS by creating hundreds of hostnames all pointing to the target's IP address.

See RFC6455 section 4.1 bullet point 2 for the exact wording of the requirements.

Once a socket has been used as a WebSocket, it cannot be reused either as a WebSocket or as an HTTP socket.

## Challenges

### Proxy lookup

The decision of whether or not to throttle must be made after any proxy scripts have been run, since the answer will be different if we are going through a proxy. This means the simple approach of doing the throttling in the WebSocket code before even creating the URLRequest will not work.

### Hostname lookup

The existing (old) implementation actually throttles to all IPs that a host resolves to, not just the one we actually connect to. This is implemented in `net::WebSocketThrottle`. It opens up some boring DoS attacks.

Example of boring DoS attack

goodguy.example.com resolves to 8.8.8.8

badguy.example.com resolves to 223.0.0.1, 223.0.0.2, 223.0.0.3, 223.0.0.4, …, 8.8.8.8

goodguy.example.com has a useful WebSocket service running on 8.8.8.8:443. badguy hacks an ad network to deliver Javascript which repeatedly attempts to connect to wss://badguy.example.com/. Most of the time it hits one of the blackhole addresses, taking some minutes to timeout. Because both resolve to 8.8.8.8, connections to wss://goodguy.example.com/ must wait in the queue behind connections to wss://badguy.example.com/

## Proxies

Chrome has a global limit of connections per-proxy. Exceeding the limit is not practical or desirable, but it is an open question whether WebSockets should use every available socket or leave some for HTTP. See http://crbug.com/347488.

When the proxy limit is reached, HTTP(S) attempts to close any idle connections, then queues connections waiting for an available slot. It might be more developer-friendly for WebSocket connections to fail in this condition, since it provides more feedback. However, it is not clear that this is permitted by RFC6455.

WebSocket uses a separate pool of proxy connections from HTTP, which could mean we make up to twice as many proxy connections as we are supposed to. That would be bad.

We have no automated integration tests for WebSocket through proxies. Manual testing environments for proxies are laborious to set up.

## “Happy Eyeballs”

Where a host has both IPv6 and IPv4 addresses, Chrome will start a connection to an IPv6 address, then if there hasn’t been a response after a short period, start another connection to an IPv4. The first of the two connections to actually complete will be used. This is needed to deal with the abundance of hosts with broken IPv6 connectivity. This behaviour is quite important for a good user experience and should be retained for WebSockets if possible.

## Backup jobs

`net::ClientSocketPoolBaseHelper` will start a second connect job if the group is empty and the first one doesn’t succeed within a timeout. See https://code.google.com/p/chromium/codesearch#chromium/src/net/socket/client_socket_pool_base.cc&l=411

This is probably harmless.

# Proposed Design

## High-level Overview

Although the wording of RFC6455 treats the proxy case as a special-case of the direct connection case, for Chrome purposes it is easier to treat it as a completely separate case.

| | **Direct WebSocket** | **WebSocket over proxy** | **HTTP (for comparison)** |
|---|---|---|---|
| Number of connections allowed in CONNECTING[1] state. | 1 | 1 | 6 |
| Number of connections allowed in CONNECTED state. | ∞ | ∞ | 6 |
| Unit of throttling | ip:port | hostname:port | hostname:port |
| Global limit on number of connections in CONNECTING state. | ∞ | “a reasonably low number” | ∞ |

# Design for Direct Connection Case

“Direct” means either plain TCP/IP, or TLS. It includes the cases where there are middleboxes performing NAT or transparent proxying.

IP lookups are performed in the `net::TransportConnectJob` class. It is the first place where the information necessary to perform throttling is available. Normally it passes a list of addresses to `ClientSocketFactory::CreateTransportClientSocket` but it is possible to filter the list to pass a single address at a time. This makes it a possible place to insert a decision to throttle. To avoid interfering with existing code, we need a new class, `net::WebSocketTransportConnectJob` which splits up the addresses returned by the resolver and performs throttling based on IPEndPoint if needed. We also need a new

[1] The socket is “CONNECTING” from the start of the handshake until it either completes successfully, or fails. See RFC6455 Section 4.1.

`net::WebSocketTransportClientSocketPool` which ensures that endpoints are unlocked when sockets are returned to the pool.

# Changes To Existing Classes

`net::ClientSocketPoolManagerImpl` already has code in the constructor to vary the arguments to `net::TransportClientSocketPool` depending on whether it is for WebSockets or not. It will need to be modified to create a `net::WebSocketTransportClientSocketPool` for WebSocket use.

A call will be added from `net::WebSocketBasicHandshakeStream::Upgrade()` to `WebSocketTransportClientSocketPool::UnlockEndpoint()` to release the lock on the endpoint and allow other connections to proceed.

# New Classes

`net::WebSocketTransportClientSocketPool` will be a subclass of `net::TransportClientSocketPool`. It will re-implement all of the virtual methods without delegating to `net::ClientSocketPoolBase`. For simplicity, it will not retain idle sockets, at least not in the first version. This will avoid the need to duplicate most of the logic of `net::ClientSocketPoolBase`.

`net::WebSocketTransportConnectJob` will take care of IP address resolution, IPEndPoint throttling, and connecting sockets.

## SubJob

In the existing `net::TransportConnectJob`, racing of IPv6 connections with IPv4 connections is handled separately from the main state machine. Because in the WebSocket case the state machine needs more states to handle connections which are throttled, this approach becomes unreasonably complicated.

Instead, `WebSocketTransportConnectJob` will separate the returned addresses into two lists, IPv6 addresses and IPv4 addresses. It will then create one `WebSocketTransportConnectJob::SubJob` for each of the two lists, and race them. As with `TransportConnectJob`, the IPv6 addresses will be given a 0.3 second head-start[2], and the first `SubJob` to return a connected socket wins the race.

This will give slightly different semantics in edge cases, but should make little practical difference.

`WebSocketTransportConnectJob::SubJob` further divides the list it is given into individual endpoints. Each address is first checked with the `WebSocketEndPointLockManager` to see whether it has already been claimed by a different connect job. If it has, the `SubJob` will be

[2] If there are no IPv6 addresses, the IPv4 connections start immediately. If all the IPv6 addresses fail in less than 0.3 seconds, the IPv4 connections will start at that point without waiting for the timer to expire.

added to a queue and wait until all of the connect jobs ahead of it in the queue are complete. For this purpose, `SubJob` subclasses `base::LinkNode`. Once the `SubJob` is the "owner" of the `IPEndPoint`, it proceeds to use `CreateTransportClientSocket(),` passing in an address list containing only one endpoint.

Normally, iteration through the endpoints returned by DNS lookup is performed by `TCPClientSocket`; however, since `SubJob` will only try one endpoint at a time, it needs to handle the looping through the address list itself. In particular, when a connection fails, `TransportConnectJob` normally passes the error back immediately, whereas `WebSocketTransportConnectJob` should only do that if there are no more addresses in the list to try.

# Issues

The major issue with this design as it stands is that proxy connections for WebSockets share the `ConnectionPoolManager` with direct connections. However, there are various conceivable workarounds for this. In the end, WebSockets cannot be taking proxy connections from a separate pool to HTTP connections because the proxy itself does not have a separate pool of resources for serving WebSocket connections.

There is a per-proxy limit of 32 connections by default; it can be overridden by setting the MaxConnectionsPerProxy administrator policy. This is anecdotally widely set to a lower value in corporate environments with underpowered shared proxies. Because there is a separate instance of `net::ClientSocketPoolManagerImpl` for WebSockets, the proxy server can end up with MaxConnectionsPerProxy × 2 connections, which would probably be considered unexpected behaviour.

For these reasons, it would probably be preferable to use the HTTP instantiations of `TransportClientSocketPool`, `HttpProxyClientSocketPool` and `SOCKSClientSocketPool` when a proxy is in use.

Another concern is how the `WebSocketTransportClientSocketPool`, which ignores group limits, will interact with higher-level pools that enforce them. Hopefully being the lowest-level pool and not keeping idle connections around will avoid major problems.

Currently when the experimental TCP FastOpen flag is enabled, and DNS returns multiple IPs for a host, Chrome will fail to fall back from a non-working IP to a working IP. This issue is common with HTTP, but because `WebSocketTransportClientSocketPool` duplicates the address selection logic from `TCPClientSocket`, any fix will have to be duplicated too. Similarly, the "Happy Eyeballs" fix for broken IPv6 connectivity will fail with TCP FastOpen enabled.

# Discussion

# Alternatives Considered

## Alternatives to creating WebSocketTransportClientSocketPool

- The logic for WebSockets could go in the ordinary `TransportClientSocketPool`, controlled by a runtime flag. However, this would lead to increased complexity and maintainability issues. As a general principle, knowledge of WebSockets should not be required for general maintenance of the HTTP stack.
- A `WebSocketTransportConnectJob` could be injected into a `TransportClientSocketPool` object. This doesn't work because the pool itself needs to contain logic to unlock endpoints when sockets are released.
- Logic could be added to `TCPClientSocket` to optionally implement WebSocket throttling semantics. This has the benefit of simplicity; the disadvantage is that it is intrusive and everybody hates the idea.
- As mentioned above, the old implementation throttles all IP address for the hostname. This is complex because the sets of IP addresses for different hostnames can overlap in arbitrary ways, resulting in a queue that is not strictly FIFO. Bad experience with this approach was the major impetus to try to do something better.
- WebSockets could use a subclass or parallel implementation of `TCPClientSocket` to implement the throttling functionality. This would involve copying most of the code from `TCPClientSocket`. The copied code would need to be kept up-to-date, which would be a maintenance burden. Two alternatives were considered to avoid copying code:
    - `TCPClientSocket` and `TCPIPEndPointThrottledClientSocket` could both delegate most of their functionality to a helper object, with only the parts that need to be different living in the exposed class. This approach was discarded due to a sense that we have too many levels of abstraction already.
    - `TCPClientSocket` could have a state machine object injected into it at construction time, to which it delegates decisions about how to respond to various events. Rejected due to complexity.
- `TCPClientSocket` could expose its IP selection logic to the caller, allowing the caller to suspend the connection after an IP address is selected. Rejected due to complexity.
- IP selection logic could be removed from `TCPClientSocket` completely, and performed by the calling class. Rejected due to the size and disruptive nature of the changes needed.
- Logic could be placed in `TCPSocket` instead. This is a lower-level platform-specific class; changing it would be riskier and require more changes.

## Alternatives to subclassing TransportClientSocketPool

- Subclassing the concrete implementation class `TransportClientSocketPool` to create `WebSocketTransportClientSocketPool` is arguably the ugliest part of this design. Unfortunately, client classes cannot simply use the `ClientSocketPool` interface

because the type of the `|params|` parameter passed to methods `RequestSocket()` and `RequestSockets()` depends upon the concrete type `TransportClientSocketPool`.

- `TransportClientSocketPool` itself could be changed to an interface type, with the implementation moved to `TransportClientSocketPoolImpl` (for example). However, this would add an extra layer of abstraction, making the code harder to follow while providing no benefit to HTTP.
- The HTTP stack could be taught to understand `WebSocketTransportClientPool` natively. This would be very invasive and defeat our design goal to be minimally intrusive.

## Why not delegate to net::ClientSocketPoolBase?

- `ClientSocketPoolBase` will sometimes delete `StreamSocket` objects without providing any hook for us to remove associated endpoint locks. One case is where a request is cancelled before the connection completes. Adding hooks everywhere `ClientSocketPoolBase` deletes a `StreamSocket` object would be intrusive.
- Wrapping `StreamSockets` in a delegating type in order to hook the destructor would double the virtual call overhead on `Read()` and `Write()`, and is ugly.
- Putting something in the base `StreamSocket` interface destructor to hook destruction would be evil and wrong.
- Most of the functionality of `ClientSocketPoolBase` simply isn't needed for WebSockets anyway.

## Code Locations

- net/websockets
- net/socket

## Design for Proxy case

TBD.

## Revision History

- 2014-02-28: First version.
- 2014-03-18: Updated with more notes.
- 2014-03-25: Direct implementation details.
- 2014-03-28: Document the IPEndPointThrottler::Release() method that eliminates the necessity to change the StreamSocket interface.
- 2014-03-28: Removed the language about preconnects being harmful; tyoshino@chromium.org ruled that they are acceptable in moderation.
- 2014-04-25: Considerably rewritten to reflect the WebSocketTransportClientSocketPool-based implementation.