Client Lifecycle

An Undici [Client]() can be best described as a state machine. The following list is a summary of the various state transitions the `Client` will go through in its lifecycle. This document also contains detailed breakdowns of each state.

> *This diagram is not a perfect representation of the undici Client. Since the Client class is not actually implemented as a state-machine, actual execution may deviate slightly from what is described below. Consider this as a general resource for understanding the inner workings of the Undici client rather than some kind of formal specification.*

State Transition Overview

- A `Client` begins in the **idle** state with no socket connection and no requests in queue.
  - The *connect* event transitions the `Client` to the **pending** state where requests can be queued prior to processing.
  - The *close* and *destroy* events transition the `Client` to the **destroyed** state. Since there are no requests in the queue, the *close* event immediately transitions to the **destroyed** state.
- The **pending** state indicates the underlying socket connection has been successfully established and requests are queueing.
  - The *process* event transitions the `Client` to the **processing** state where requests are processed.
  - If requests are queued, the *close* event transitions to the **processing** state; otherwise, it transitions to the **destroyed** state.
  - The *destroy* event transitions to the **destroyed** state.
- The **processing** state initializes to the **processing.running** state.
  - If the current request requires draining, the *needDrain* event transitions the `Client` into the **processing.busy** state which will return to the **processing.running** state with the *drainComplete* event.
  - After all queued requests are completed, the *keepalive* event transitions the `Client` back to the **pending** state. If no requests are queued during the timeout, the **close** event transitions the `Client` to the **destroyed** state.
  - If the *close* event is fired while the `Client` still has queued requests, the `Client` transitions to the **process.closing** state where it will complete all existing requests before firing the *done* event.
  - The *done* event gracefully transitions the `Client` to the **destroyed** state.
  - At any point in time, the *destroy* event will transition the `Client` from the **processing** state to the **destroyed** state, destroying any queued requests.
- The **destroyed** state is a final state and the `Client` is no longer functional.

A state diagram representing an Undici Client instance

> *The diagram was generated using Mermaid.js Live Editor. Modify the state diagram [here]()*

State details

idle

The **idle** state is the initial state of a `Client` instance. While an `origin` is required for instantiating a `Client` instance, the underlying socket connection will not be established until a request is queued using [`Client.dispatch()`](). By calling `Client.dispatch()` directly or using one of the multiple implementations

(`Client.connect()` , `Client.pipeline()` , `Client.request()` , `Client.stream()` , and `Client.upgrade()`), the `Client` instance will transition from **idle** to **pending** and then most likely directly to **processing**.

Calling `Client.close()` or `Client.destroy()` transitions directly to the **destroyed** state since the `Client` instance will have no queued requests in this state.

pending

The **pending** state signifies a non-processing `Client`. Upon entering this state, the `Client` establishes a socket connection and emits the `'connect'` event signalling a connection was successfully established with the `origin` provided during `Client` instantiation. The internal queue is initially empty, and requests can start queueing.

Calling `Client.close()` with queued requests, transitions the `Client` to the **processing** state. Without queued requests, it transitions to the **destroyed** state.

Calling `Client.destroy()` transitions directly to the **destroyed** state regardless of existing requests.

processing

The **processing** state is a state machine within itself. It initializes to the **processing.running** state. The `Client.dispatch()` , `Client.close()` , and `Client.destroy()` can be called at any time while the `Client` is in this state. `Client.dispatch()` will add more requests to the queue while existing requests continue to be processed. `Client.close()` will transition to the **processing.closing** state. And `Client.destroy()` will transition to **destroyed**.

running

In the **processing.running** sub-state, queued requests are being processed in a FIFO order. If a request body requires draining, the *needDrain* event transitions to the **processing.busy** sub-state. The *close* event transitions the Client to the **process.closing** sub-state. If all queued requests are processed and neither `Client.close()` nor `Client.destroy()` are called, then the **processing** machine will trigger a *keepalive* event transitioning the `Client` back to the **pending** state. During this time, the `Client` is waiting for the socket connection to timeout, and once it does, it triggers the *timeout* event and transitions to the **idle** state.

busy

This sub-state is only entered when a request body is an instance of Stream and requires draining. The `Client` cannot process additional requests while in this state and must wait until the currently processing request body is completely drained before transitioning back to **processing.running**.

closing

This sub-state is only entered when a `Client` instance has queued requests and the `Client.close()` method is called. In this state, the `Client` instance continues to process requests as usual, with the one exception that no additional requests can be queued. Once all of the queued requests are processed, the `Client` will trigger the *done* event gracefully entering the **destroyed** state without an error.

destroyed

The **destroyed** state is a final state for the `Client` instance. Once in this state, a `Client` is nonfunctional. Calling any other `Client` methods will result in an `ClientDestroyedError`.