

KIP-601: Configurable socket connection timeout in NetworkClient


- [Status](#)
- [Motivation](#)
- [Public Interfaces](#)
- [Proposed Changes](#)
 - [NetworkClient](#)
 - [ClusterConnectionStates](#)
 - [When would the connection timeout increase?](#)
 - [Relationship between the proposed connection timeout, existing request timeout, and existing API timeout](#)
 - [Connection timeout dominates both request timeout and API timeout](#)
 - [Neither request timeout or API timeout dominates connection timeout](#)
- [Compatibility, Deprecation, and Migration Plan](#)
- [Rejected Alternatives](#)

Status

Current state: Accepted

Discussion thread: [here](#)

JIRA:

 Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Currently, the socket connection timeout is depending on system setting `tcp_sync_retries`. The timeout value is $2^{(\text{tcp_sync_retries} + 1)} - 1$ seconds. For the reasons below, we want to control the client-side socket timeout directly using configuration files.

1. The default value of `tcp_sync_retries` may be large, such as 6 for Linux. It means the default timeout value is 127 seconds for finishing the three-way handshake. A shorter timeout at the transportation level will help clients detect dead nodes faster. The existing configuration "`request.timeout.ms`" sets an upper-bound of the time used by both the transportation and application layer whose complexity varies. It's risky to lower "`request.timeout.ms`" for detecting dead nodes quicker because of the involvement of the application layer logic.
2. The existing configuration "`request.timeout.ms`" is not able to time out the connections properly. Because "`request.timeout.ms`" only affects in-flight requests, the connecting node won't get disconnected after "`request.timeout.ms`" hits, which may cause side effects. For example, the `leastLoadedNode()` provides a cached node with the criteria below.
 - a. Provide the connected node with least number of inflight requests
 - b. If no connected node exists, provide the connecting node with the largest index in the cached list of nodes.
 - c. If no connected or connecting node exists, provide the disconnected node which respects the reconnect backoff with the largest index in the cached list of nodes.

A node will remain the "connecting" status until $2^{(\text{tcp_sync_retries} + 1)} - 1$ seconds elapsed, even if the requests binding to this node timed out. So the `leastLoadedNode()` might keep providing this same node and other nodes won't get a chance to process any requests. For example, when the user specifies a list of N bootstrap-servers and no connection has been built between the client and the servers, the least loaded node provider will poll all the server nodes specified by the user. If M servers in the bootstrap-servers list are offline, the client may take $(127 * M)$ seconds to connect to the cluster. In the worst case when $M = N - 1$, the wait time can be several minutes.

Considering the potential approval of KIP-612 which proposes to throttle connection setup, we propose an exponential connection setup timeout to help the NetworkClient

1. Detect the dead node faster and try the request on other nodes if applicable
2. Be able to wait longer and longer for finishing the connection if the broker side connection setup is throttled.

Public Interfaces

We propose two new common client configs

socket.connection.setup.timeout.ms: The amount of time the client will wait for the initial socket connection to be built. If the connection is not built before the timeout elapses the network client will close the socket channel. The default value will be 10 seconds.

socket.connection.setup.timeout.max.ms: The maximum amount of time the client will wait for the initial socket connection to be built. The connection setup timeout will increase exponentially for each consecutive connection failure up to this maximum. To avoid connection storms, a randomization factor of 0.2 will be applied to the backoff resulting in a random range between 20% below and 20% above the computed value. The default value will be 127 seconds.

The formula to calculate the latest connection setup timeout is as follows, where the random factor is to prevent connection storms:

```
MIN(socket.connection.setup.timeout.max.ms, socket.connection.setup.timeout.ms * 2 ^ (failures - 1) * random(0.8, 1.2))
```

Proposed Changes

NetworkClient

1. The new config will be a common client config. The **NetworkClient** will keep the proposed configs as new properties.
2. **NetworkClient.poll()** will iterate all connecting nodes and disconnect those timed out connections using the exact approach as it handles "request.timeout.ms"
3. The node providing criteria **C** in the **leastLoadedNode()** will also change accordingly. Now the criteria should look like below:
 - a. Provide the connected node with least number of inflight requests
 - b. If no connected node exists, provide the connecting node with the largest index in the cached list of nodes.
 - c. If no connected or connecting node exists, provide the disconnected node which respects the reconnect backoff and is least recently provided. Consider the case when we have multiple **DISCONNECTED** nodes and the time interval between the two **provide()** invokes is greater than **reconnect.backoff.ms**. The Provider can provide the same nodes all the time. Thus, the provider should provide the least recently provided nodes among all nodes passing the **canConnect()** check.

ClusterConnectionStates

1. Add a new **HashSet** property **ConnectingNodes** keeping all the connecting node ids.
2. Will expose a public API that returns the **ConnectingNodes** mentioned in #1, helping the **NetworkClient** process the timeout iteration.
3. State transition:
 - a. **ClusterConnectionStates.connecting()** will add the node id to **ConnectingNodes**
 - b. **ClusterConnectionStates.ready()** will remove the node id to **ConnectingNodes**
 - c. **ClusterConnectionStates.disconnected()** will remove the node id from **ConnectingNodes**

When would the connection timeout increase?

Every time the timeout hits, the timeout value of the next connection try will increase. The timeout will hit iff a connection stays at the **`connecting`** state longer than the timeout value, as indicated by **ClusterConnectionStates.NodeConnectionState**. The connection state of a node will change iff **SelectionKey.OP_CONNECT** is detected by **nioSelector.Select()**. The connection state may transit from **`connecting`** to

1. **`disconnected`** when **SocketChannel.finishConnect()** throws **IOException**.
2. **`connected`** when **SocketChannel.finishConnect()** return **TRUE**.

In other words, the timeout will hit and increase iff the interested **SelectionKey.OP_CONNECT** doesn't happen before the timeout arrives, which means, for example, network congestion, failure of the ARP request, packet filtering, routing error, or a silent discard may happen.

Relationship between the proposed connection timeout, existing request timeout, and existing API timeout

Connection timeout dominates both request timeout and API timeout

When connection timeout hits, the connection will be closed. The client will be notified either by the responses constructed by **NetworkClient** or the callbacks attached to the request. As a result, the request failure will be handled before either connection timeout or API timeout arrives.

Neither request timeout or API timeout dominates connection timeout

Request timeout: Because request timeout only affects in-flight requests, after the API **NetworkClient.ready()** is invoked, the connection won't get closed after "request.timeout.ms" hits. Before

1. the **SocketChannel** is connected
2. **SSL** handshake finished
3. authentication has finished (**SASL**)

, clients won't be able to invoke **NetworkClient.send()** to send any request, which means no in-flight request targeting to the connection will be added.

API timeout: In **AdminClient**, API timeout acts by putting a smaller and smaller timeout value to the chain of requests in a same API. After the API timeout hits, the retry logic won't close any connection. In **consumer**, API timeout acts as a whole by putting a limit to the code block executing time. The retry logic won't close any connection as well.

Compatibility, Deprecation, and Migration Plan

No impact

Rejected Alternatives

1. Use **request.timeout.ms** to time out the socket connection at the client level instead of the network client level
 - a. **request.timeout.ms** is at the client/request level. We need one in the **NetworkClient** level to control the connection states.
 - b. The transportation layer timeout should be relatively shorter than the request timeout. It's good to have a separate config.
 - c. In some scenarios, **request.timeout.ms** is not able to time out the connections properly.
2. Use the number of failed attempts as the prioritizing rules to choose between disconnected nodes in **leastLoadedNode()** when no connected or connecting node exists.
 - a. "For example, if a new node joins the cluster, it will have 0 failed connect attempts, whereas the existing nodes will probably have more than 0. So all the clients will ignore every other node and pile on to the new one." CR to [Colin McCabe](#)
3. Add a new connection state **TIMEOUT** besides **DISCONNECTED**, **CONNECTING**, **CHECKING_API_VERSIONS**, **READY**, and **AUTHENTICATING_FAILED**.
 - a. We don't necessarily need to differentiate the timeout and disconnected states.
4. a lazy socket connection time out. That is, the **NetworkClient** will only check and disconnect timeout connections in **leastLoadedNode()**.

Pros:

 - a. Usually, when clients send a request, they will ask the network client to send the request to a specific node. In these cases, the **connection.setup.timeout** won't matter too much because the client doesn't want to try other nodes for that specific request. The request level timeout would be enough. The metadata fetcher fetches the status of the nodes periodically so the clients will reassign the timeout request correspondingly to a different node.
 - b. Consumer, producer, and **AdminClient** are all using **leastLoadedNode()** for metadata fetches, where the connection setup timeout can play an important role. Unlike other requests can refer to the metadata for node condition, the metadata requests can only blindly choose a node for retry in the worst scenario. We want to make sure the client can get the metadata smoothly and as soon as possible. As a result, we need this **socket.connection.setup.timeout.ms**.
 - c. Implementing the timeout in **NetworkClient.poll()** or anywhere else might need an extra iteration of all nodes, which might downgrade the network client performance.
 - d. Clients fetch metadata periodically which means **leastLoadedNode()** will get called frequently. So the timeout checking frequency is guaranteed.

However, we need a more common and universal timeout for all connections. New scenarios may jump out beside the current metadata fetching scenario