Leases

Table of contents

> ⚠ **Attention**
>
> NVIDIA Triton Management Service (TMS) will reach the end of life on July 31, 2024. The version 1.4.0 is the last release.

Overview

A lease is the primary organizational unit used by TMS. Leases allow users to describe which models to load, as well as control their lifecycle. Leases provide a convenient means of describing the Triton instance where the models should be loaded, without having to manage the Triton instance directly. Additionally, leases allow you to configure features like autoscaling or sharing Triton servers.

A lease consists of the following information:

- A model or ensemble of models that will be loaded together in a Triton instance.
- A description of the Triton instance where to load the models. This can be either:
    - A bespoke Triton instance which will be created just for this lease and will not be shared. Bespoke Triton instances support autoscaling.
    - The name of a pre-existing [Triton pool](./triton-pools.md], where the lease may share a Triton instance with other leases.
- Information about the duration of the lease, including whether it supports automatic renewal based on usage.

For example, a simple lease might have the following characteristics:

- It consists of a single model named `model_A`, along with the URI from which to fetch it.
- Uses a bespoke Triton instance with the default configuration as set by the system administrator.

- Lasts for the default duration as set by the system administrator.

A more complex lease might look as follows:

- It consists of an ensemble of models. The models are named `model_A`, `model_B`, and `model_C`, and each is specified along with the URI from which to fetch it. These models will be loaded in the specified order to ensure the ensemble works properly.
- It describes a bespoke Triton instance on which to run with custom resource requirements:
    - The Triton instance supports autoscaling up to four copies, scaling up when inference queue time exceeds 200ms.
    - Each Triton requires 2 GPUs, 4 CPUs, and 32 GB of main memory.
- The lease should remain active for 8 hours, and automatically renew for another four hours so long as it has served inference requests in the 30 minutes before it would expire.

The exact lifecycle of a lease will vary depending on the application requirements, but they will all follow this general outline:

- Create a lease via a `Lease.Acquire()` API call and get the URL of the Triton instance where the models were loaded.
- Run inference against the models in the lease using the Triton inference API.
- Potentially renew the lease via `Lease.Rewnew()` calls, or let it automatically renew if it is configured to renew while still being used.
- Either manually release the lease via a `Lease.Release()` call, or let it expire. Either way, this will either free the associated Triton instance for bespoke leases or mark the resources as available for leases running on pooled Triton instances.

For more details on the available operations, see the section below. There is also a tutorial that will guide you through the basics of of working with leases.

Lease Operations

The Lease Service exposes a number of RPC end-points: `Acquire`, `Release`, `Renew`, and `Status`. The Triton Allowlist Service exposes the following RPC end-points: `Append`, `List` and `Remove`. Each of the end-point accept a single structured request and respond with a structured response.

> **Note**
>
> The gRPC protocol supports streaming requests and/or responses. This means that one or both sides of the interaction can stream data to the other. Functionally, this allows the server to being sending response data before the client has finished sending request data.

The expected order of operations with regards to the Lease Service are as follows:

1. `Lease/Acquire` to create a new lease with a specified set of AI models.

   Assuming the request is successful, the response will include a unique identifier and an expiration date for the new lease.

   All models in a lease acquire request are considered bundled. They cannot be loaded or unloaded separately. Additionally, all models in a lease will be loaded into the same instance of Triton Inference Server. If it is impossible to do so (e.g. insufficient memory), then the lease will be marked as invalid and any models successfully loaded models will be unloaded after the first model load failure is detected. TMS does not support partially loaded leases.

   A lease can created as part of a Triton Pool or using a bespoke Triton instance. This is determined by the use of the `triton_options` value in the gRPC API.

   This RPC begins streaming a response once the request has been received. The server will send a series of model status updates to the caller to show continued progress as the lease's models are deployed. Model status updates will be sparse (not include status of every model every time).

   The final response from the server will include status for every model in the request as well as data for the lease itself.

2. `Lease/Renew` to extend the lease's duration. Once a lease is renewed it assigned a new expiration date.

   Once a lease has expired, it is no longer valid and any associated models will be unloaded and become unavailable. Any resources consumed by the lease are returned to the hosting Triton Inference Server to be used by future leases. In the case that a Triton Inference Server instance becomes unnecessary, it will be deleted and its resources returned to the cluster.

3. `Lease/Status` to get the current status of a specific lease.

   Requesting the status of an expired or released lease is a valid operation.

4. `Lease/Release` to terminate a lease before its expiration is reached.

   Once a lease has been released, it is no longer valid and any associated models will be unloaded and become unavailable. Any resources consumed by the lease are returned to the hosting Triton Inference Server to be used by future leases. In the case that a Triton Inference Server instance becomes unnecessary, it will be deleted and its resources returned to the cluster.