/\* Copyright 2016 The TensorFlow Authors. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License");

you may not use this file except in compliance with the License.

You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

==============================================================================\*/

syntax = "proto3";

package tensorflow.grpc;

import "tensorflow/core/protobuf/master.proto";

option java\_outer\_classname = "MasterServiceProtos";

option java\_multiple\_files = true;

option java\_package = "org.tensorflow.distruntime";

option go\_package = "github.com/tensorflow/tensorflow/tensorflow/go/core/protobuf/for\_core\_protos\_go\_proto";

////////////////////////////////////////////////////////////////////////////////

//

// MasterService defines a TensorFlow service with which a client can

// interact to execute a distributed TensorFlow computation.

//

// A master service keeps track of multiple "master sessions". Each

// session encapsulates a computation graph and its associated state,

// and typically corresponds to a single "client session" (e.g. a

// `tensorflow::Session` instance).

//

// A session is responsible for the following:

// \* assigning each node to a device (locally or remotely) using a

// placement algorithm. This may make decisions based on collected

// statistics from the workers in the system (e.g., memory usage,

// bandwidth consumption, etc.)

//

// \* inserting intermediate nodes and edges to support cross-device

// and cross-process data flows and resource management.

//

// \* issuing commands to workers to execute the subgraphs associated

// with those workers.

//

// Typically, a client carries out an iterative computation

// (e.g. training) by invoking RPCs against the master in a

// client-side loop. The client first creates a client session that

// connects to a particular master (using gRPC for example). The

// master creates a corresponding master session that is hosted on

// the master and caches state between the client's invocations.

//

// After the session is established, the master returns an opaque

// handle to the client that can be used to associate the client and

// master sessions.

//

// The client may send an initial graph to the master in the

// CreateSession call, and add nodes to the graph using ExtendSession.

//

// The most frequent operation a master is "RunStep", which implements

// the `Session::Run()` API. It supports feeding in arguments,

// executing a dataflow computation, and fetching arguments.

//

// Finally, when the client no longer needs the session, it should

// close the session by invoking CloseSession, which allows the master

// to reclaim resources associated with the session. The master may

// implement a garbage collection scheme that closes sessions that

// have been inactive for some time.

//

// For example, the following pseudo-code illustrates how a client

// interacts with a master:

//

// stub = NewStub("/job:mnist/replica:0/task:0")

// {handle} = stub->CreateSession({graph\_def})

// do {

// stub->RunStep({handle, {feeds}, {fetches}})

// // The client can evaluate a predicate locally, based on the

// // result of `fetches`, to determine whether to terminate. For

// // example, it might fetch the loss and evaluate whether it is less

// // than some threshold.

// } while (!should\_stop({fetches}));

// stub->CloseSession({handle})

//

////////////////////////////////////////////////////////////////////////////////

service MasterService {

// Creates a session.

rpc CreateSession(CreateSessionRequest) returns (CreateSessionResponse);

// Extends a session.

rpc ExtendSession(ExtendSessionRequest) returns (ExtendSessionResponse);

// Prepares future partial run calls.

rpc PartialRunSetup(PartialRunSetupRequest) returns (PartialRunSetupResponse);

// Drives the graph computation.

rpc RunStep(RunStepRequest) returns (RunStepResponse);

// Closes a session.

rpc CloseSession(CloseSessionRequest) returns (CloseSessionResponse);

// List the devices usable by the master.

rpc ListDevices(ListDevicesRequest) returns (ListDevicesResponse);

// Close and abandon all existing sessions. Ongoing computations

// will no longer affect fresh ones via the resources in containers listed in

// the ResetRequest. See ResetRequest for more details.

rpc Reset(ResetRequest) returns (ResetResponse);

// Registers a callable for execution with RunCallable.

rpc MakeCallable(MakeCallableRequest) returns (MakeCallableResponse);

// Executes a callable registered with MakeCallable.

rpc RunCallable(RunCallableRequest) returns (RunCallableResponse);

// Frees resources associated with a callable registered with MakeCallable.

rpc ReleaseCallable(ReleaseCallableRequest) returns (ReleaseCallableResponse);

}