syntax = "proto3";

package tensorflow.eager;

import "tensorflow/core/framework/attr\_value.proto";

import "tensorflow/core/framework/device\_attributes.proto";

import "tensorflow/core/framework/function.proto";

import "tensorflow/core/framework/tensor.proto";

import "tensorflow/core/framework/tensor\_shape.proto";

import "tensorflow/core/framework/versions.proto";

import "tensorflow/core/protobuf/remote\_tensor\_handle.proto";

import "tensorflow/core/protobuf/tensorflow\_server.proto";

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

// A proto representation of an eager operation.

message Operation {

// A unique identifier for the operation. Set by the client so that the client

// can uniquely identify the outputs of the scheduled operation.

//

// In the initial implementation, sending duplicate IDs has undefined

// behaviour, but additional constraints may be placed upon this in the

// future.

int64 id = 1;

string name = 2;

message Input {

oneof item {

RemoteTensorHandle remote\_handle = 1;

TensorProto tensor = 2;

}

}

repeated Input op\_inputs = 10;

// Control Operation IDs that will be respected when ops are re-ordered by

// async execution. If async execution (+ op re-ordering) is not enabled, this

// should have no effect.

repeated int64 control\_op\_ids = 4;

map<string, AttrValue> attrs = 5;

string device = 6;

// Indicates whether the op is a component of a multi-device function.

bool is\_component\_function = 7;

// Set when is\_component\_function is true. It's initially generated

// when we create an FunctionLibraryRuntime::Options (negative value) and used

// to create Rendezvous for function execution. All components of a

// multi-device function should use the same step id to make sure that they

// can communicate through Send/Recv ops.

int64 func\_step\_id = 8;

// Indicates whether the op is a function.

bool is\_function = 9;

reserved 3;

}

message QueueItem {

// The remote executor should be able to handle either executing ops directly,

// or releasing any unused tensor handles, since the tensor lifetime is

// maintained by the client.

oneof item {

RemoteTensorHandle handle\_to\_decref = 1;

Operation operation = 2;

SendTensorOp send\_tensor = 3;

// Takes a FunctionDef and makes it enqueable on the remote worker.

RegisterFunctionOp register\_function = 4;

CleanupFunctionOp cleanup\_function = 5;

// A remote executor is created to execute ops/functions asynchronously

// enqueued in streaming call. Request with this item type waits for pending

// nodes to finish on the remote executor and report status.

SyncRemoteExecutorForStream sync\_remote\_executor\_for\_stream = 6;

SendPackedHandleOp send\_packed\_handle = 7;

}

}

message QueueResponse {

// `shape` and `tensor` cannot be set in the same response.

// Shapes of output tensors for creating remote TensorHandles.

repeated TensorShapeProto shape = 1;

// Optional. If set, represents the output devices of a function.

repeated string device = 3;

// Output tensors of a remote function. Set when Operation.id is invalid.

repeated TensorProto tensor = 2;

}

message CreateContextRequest {

// Identifies the full cluster, and this particular worker's position within.

ServerDef server\_def = 1;

// Whether the ops on the worker should be executed synchronously or

// asynchronously. By default, ops are executed synchronously.

bool async = 2;

// Number of seconds to keep the context alive. If more than keep\_alive\_secs

// has passed since a particular context has been communicated with, it will

// be garbage collected.

int64 keep\_alive\_secs = 3;

// This is the version for all the ops that will be enqueued by the client.

VersionDef version\_def = 4;

// Device attributes in the cluster

repeated DeviceAttributes cluster\_device\_attributes = 6;

// The ID of the created context. This is usually a randomly generated number,

// that will be used to identify the context in future requests to the

// service. Contexts are not persisted through server restarts.

// This ID will be used for all future communications as well. It is essential

// that both ends use this ID for selecting a rendezvous to get everything to

// match.

fixed64 context\_id = 7;

// The view ID of the context.

fixed64 context\_view\_id = 8;

// For a multi device function, if false, eagerly copy all remote inputs to

// the default function device; if true, lazily copy remote inputs to their

// target devices after function instantiation to avoid redundant copies.

bool lazy\_copy\_remote\_function\_inputs = 9;

reserved 5;

}

message CreateContextResponse {

// List of devices that are locally accessible to the worker.

repeated DeviceAttributes device\_attributes = 2;

reserved 1;

}

message UpdateContextRequest {

// Identifies the full cluster, and this particular worker's position within.

ServerDef server\_def = 1;

// Device attributes in the cluster.

// If this field is empty, it indicates that this is a simple update request

// that only increments the cluster view ID and does not require changes to

// the workers it connects to.

repeated DeviceAttributes cluster\_device\_attributes = 2;

// The ID of the context to be updated. A context with the specified ID must

// already exist on the recepient server of this request.

fixed64 context\_id = 3;

// The view ID of the context, which should be contiguously incremented when

// updating the same context.

fixed64 context\_view\_id = 4;

}

message UpdateContextResponse {

// List of devices that are locally accessible to the worker.

repeated DeviceAttributes device\_attributes = 1;

}

message EnqueueRequest {

fixed64 context\_id = 1;

repeated QueueItem queue = 3;

}

message EnqueueResponse {

// A single operation response for every item in the request.

repeated QueueResponse queue\_response = 1;

}

message WaitQueueDoneRequest {

fixed64 context\_id = 1;

// Ids to wait on. If empty, wait on everything currently pending.

repeated int64 op\_id = 2;

}

message WaitQueueDoneResponse {

// TODO(nareshmodi): Consider adding NodeExecStats here to be able to

// propagate some stats.

}

message RunComponentFunctionRequest {

fixed64 context\_id = 1;

Operation operation = 2;

// The output indices of its parent function.

repeated int32 output\_num = 3;

}

message RunComponentFunctionResponse {

repeated TensorShapeProto shape = 1;

repeated TensorProto tensor = 2;

}

message KeepAliveRequest {

fixed64 context\_id = 1;

}

message KeepAliveResponse {

// If the requested context\_id is on the remote host, set the context view ID.

fixed64 context\_view\_id = 1;

}

message CloseContextRequest {

fixed64 context\_id = 1;

fixed64 context\_view\_id = 2;

}

message CloseContextResponse {}

message RegisterFunctionOp {

FunctionDef function\_def = 1;

// If true, it means that function\_def is produced by graph partition during

// multi-device function instantiation.

bool is\_component\_function = 2;

// All necessary FunctionDefs and GradientDefs to expand `function\_def`.

// When is\_component\_function is true, `function\_def` could be a nested

// function, since some nodes in its parent's function body could be

// replaced with a new function by the graph optimization passes. No need to

// add FunctionDefs here to the function cache in EagerContext since they

// won't be executed as KernelAndDevices.

FunctionDefLibrary library = 3;

}

// Cleanup the step state of a multi-device function (e.g. tensors buffered by

// a `Send` op but not picked up by its corresponding `Recv` op).

message CleanupFunctionOp {

int64 step\_id = 1;

}

message SyncRemoteExecutorForStream {}

message SendTensorOp {

// All remote tensors are identified by <Op ID, Output num>. To mimic this

// situation when directly sending tensors, we include an "artificial" op ID

// (which would have corresponded to the \_Recv op when not using SendTensor).

int64 op\_id = 1;

// The index within the repeated field is the output number that will help

// uniquely identify (along with the above op\_id) the particular tensor.

repeated TensorProto tensors = 2;

// The device on which the tensors should be resident.

string device\_name = 3;

}

// Send a packed TensorHandle to a remote worker.

message SendPackedHandleOp {

// Op id of the remote packed TensorHandle.

int64 op\_id = 1;

message LocalTensorHandle {

TensorProto tensor = 1;

// Device where the tensor is produced.

string device = 2;

}

message Handle {

oneof item {

LocalTensorHandle local\_handle = 1;

RemoteTensorHandle remote\_handle = 2;

}

}

repeated Handle handles = 2;

string device\_name = 3;

}

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

//

// Eager Service defines a TensorFlow service that executes operations eagerly

// on a set of local devices, on behalf of a remote Eager executor.

//

// The service impl will keep track of the various clients and devices it has

// access to and allows the client to enqueue ops on any devices that it is able

// to access and schedule data transfers from/to any of the peers.

//

// A client can generate multiple contexts to be able to independently execute

// operations, but cannot share data between the two contexts.

//

// NOTE: Even though contexts generated by clients should be independent, the

// lower level tensorflow execution engine is not, so they might share some data

// (e.g. a Device's ResourceMgr).

//

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

service EagerService {

// This initializes the worker, informing it about the other workers in the

// cluster and exchanging authentication tokens which will be used in all

// other RPCs to detect whether the worker has restarted.

rpc CreateContext(CreateContextRequest) returns (CreateContextResponse);

// This updates the eager context on an existing worker when updating the set

// of servers in a distributed eager cluster.

rpc UpdateContext(UpdateContextRequest) returns (UpdateContextResponse);

// This takes a list of Execute and DeleteTensorHandle operations and enqueues

// (in async mode) or executes (in sync mode) them on the remote server.

// All outputs of ops which were not explicitly deleted with

// DeleteTensorHandle entries will be assumed to be alive and are usable by

// future calls to Enqueue.

rpc Enqueue(EnqueueRequest) returns (EnqueueResponse);

// A streaming version of Enqueue.

// Current server implementation sends one response per received request.

// The benefit for using a streaming version is that subsequent requests

// can be sent without waiting for a response to the previous request. This

// synchronization is required in the regular Enqueue call because gRPC does

// not guarantee to preserve request order.

rpc StreamingEnqueue(stream EnqueueRequest) returns (stream EnqueueResponse);

// Takes a set of op IDs and waits until those ops are done. Returns any error

// in the stream so far.

rpc WaitQueueDone(WaitQueueDoneRequest) returns (WaitQueueDoneResponse);

// This takes an Eager operation and executes it in async mode on the remote

// server. Different from EnqueueRequest, ops/functions sent through this

// type of requests are allowed to execute in parallel and no ordering is

// preserved by RPC stream or executor.

// This request type should only be used for executing component functions.

// Ordering of component functions should be enforced by their corresponding

// main functions. The runtime ensures the following invarients for component

// functions (CFs) and their main functions (MFs):

// (1) MF1 -> MF2 ==> CF1 -> CF2 ("->" indicates order of execution);

// (2) MF1 || MF2 ==> CF1 || CF2 ("||" indicates possible parallel execution);

// (3) For CF1 and CF2 that come from the same MF, CF1 || CF2

// For executing ops/main functions, use Enqueue or StreamingEnqueue instead

// for correct ordering.

rpc RunComponentFunction(RunComponentFunctionRequest)

returns (RunComponentFunctionResponse);

// Contexts are always created with a deadline and no RPCs within a deadline

// will trigger a context garbage collection. KeepAlive calls can be used to

// delay this. It can also be used to validate the existence of a context ID

// on remote eager worker. If the context is on remote worker, return the same

// ID and the current context view ID. This is useful for checking if the

// remote worker (potentially with the same task name and hostname / port) is

// replaced with a new process.

rpc KeepAlive(KeepAliveRequest) returns (KeepAliveResponse);

// Closes the context. No calls to other methods using the existing context ID

// are valid after this.

rpc CloseContext(CloseContextRequest) returns (CloseContextResponse);

}