/\* 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;

import "google/protobuf/any.proto";

import "tensorflow/core/framework/cost\_graph.proto";

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

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

import "tensorflow/core/framework/step\_stats.proto";

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

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

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

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

import "tensorflow/core/protobuf/coordination\_config.proto";

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

import "tensorflow/core/protobuf/error\_codes.proto";

import "tensorflow/core/protobuf/named\_tensor.proto";

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

option cc\_enable\_arenas = true;

option java\_outer\_classname = "WorkerProtos";

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";

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

//

// GetStatus method request/response messages

//

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

message GetStatusRequest {}

message GetStatusResponse {

repeated DeviceAttributes device\_attributes = 1;

}

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

//

// CreateSession method request/response messages

//

// For each session,

//

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

message CreateWorkerSessionRequest {

// Sessions are identified by a given handle.

string session\_handle = 1;

// Defines the configuration of a TensorFlow worker.

ServerDef server\_def = 2;

// If true, any resources such as Variables used in the session will not be

// shared with other sessions.

bool isolate\_session\_state = 3;

// The device attributes of all the devices in the cluster.

repeated DeviceAttributes cluster\_device\_attributes = 4;

// The master task name from which the request is sent.

string master\_task = 5;

// The incarnation ID of the master task local CPU device.

// If the target worker already has a WorkerSession created previously with

// the same master task name but a different incarnation, it usually indicates

// that the previous master failed before deleting the WorkerSession on the

// worker. To prevent memory leaks, the worker should garbage collect the old

// WorkerSessions.

int64 master\_incarnation = 6;

// Configures coordination service within worker sessions.

CoordinationServiceConfig coordination\_service\_config = 7;

}

message CreateWorkerSessionResponse {}

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

//

// DeleteSession method request/response messages

//

// Deletes all worker-side state associated with the given session handle.

//

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

message DeleteWorkerSessionRequest {

// Sessions are identified by a given handle.

string session\_handle = 1;

}

message DeleteWorkerSessionResponse {}

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

//

// RegisterGraph method request/response messages

//

// For each session, after the master placed every node on a device,

// it partitions the whole graph into many subgraphs. All the nodes in

// a subgraph were in the same worker, but potentially on many devices

// owned by that worker (e.g. cpu0, plus gpu0, gpu1, ..., gpu7). The

// master registers subgraphs for a worker before running any steps. A

// successful registration returns a graph handle to be used in latter

// RunGraph requests.

//

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

message RegisterGraphRequest {

// Subgraphs are scoped within one session.

string session\_handle = 1;

// Set to true if `CreateWorkerSession` was called for `session\_handle`.

bool create\_worker\_session\_called = 6;

// "graph\_def" has the subgraph of nodes for this worker, with each node

// having its device\_name filled in.

GraphDef graph\_def = 2;

// True iff the graph (before partitioning) contains control flow nodes.

//

// As of 01/11/2015, this is no longer set by clients.

bool has\_control\_flow = 3 [deprecated = true];

// Configuration options for the session in which this graph was created.

GraphOptions graph\_options = 4;

// Field(s) used by TensorFlow Debugger (tfdbg).

DebugOptions debug\_options = 5;

// If graph\_def contains any collective ops this must be a positive

// integer used to coordinate execution with other graphs. All

// graphs in a distributed execution with the same

// collective\_graph\_key will coordinate to use the same step\_id

// concurrently so that BufRendezvous entries will make the correct

// values accessible.

int64 collective\_graph\_key = 7;

// ConfigProto from the session in which this graph was created.

// Contains additional parameters beyond graph\_options, including

// the name of the requested executor.

ConfigProto config\_proto = 8;

}

message RegisterGraphResponse {

// If the registration succeeds, returns an opaque graph\_handle to

// the master. The master calls RunGraph with graph\_handle to

// compute different steps.

string graph\_handle = 1;

}

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

//

// DeregisterGraph method request/response messages

//

// The master deregisters the given graph\_handle when the graph is no

// longer needed (e.g., the overall graph is re-scheduled and nodes

// are re-placed).

//

// The worker deregisters a graph\_handle automatically according to on

// a TTL-base policy in case of master restarts.

//

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

message DeregisterGraphRequest {

// The session\_handle used when registering the graph. If session\_handle is

// empty, a single global namespace is used.

string session\_handle = 2;

// Set to true if `CreateWorkerSession` was called for `session\_handle`.

bool create\_worker\_session\_called = 3;

// REQUIRED: graph\_handle must be returned by a RegisterGraph call

// to the same WorkerService.

string graph\_handle = 1;

}

message DeregisterGraphResponse {

// TODO(mrry): Optionally add summary stats for the graph.

}

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

//

// CleanupAll method request/response messages

//

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

message CleanupAllRequest {

// A list of container names.

//

// If 'container' is not empty, releases resources in the given

// containers in all devices.

//

// If 'container' is empty, releases resources in the default

// container in all devices.

repeated string container = 1;

}

message CleanupAllResponse {}

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

//

// RunGraph request / response messages

//

// The worker executes all subgraphs registered under graph\_handle.

// RunGraph returns after the execution finishes or an error is

// encountered.

// A sequence of RunGraphRequests with is\_partial may be sent to RunGraph for

// partial graph execution.

//

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

// Options specific to the execution of a single step.

message ExecutorOpts {

bool record\_costs = 1;

bool record\_timeline = 3;

bool record\_partition\_graphs = 4;

bool report\_tensor\_allocations\_upon\_oom = 5;

}

message RunGraphRequest {

// session\_handle is the master-generated unique id for this session.

// If session\_handle is non-empty, it must be the same as used when

// registering the graph. If it is empty, a single global namespace is used to

// search for the graph\_handle.

string session\_handle = 8;

// Set to true if `CreateWorkerSession` was called for `session\_handle`.

bool create\_worker\_session\_called = 10;

// REQUIRED: graph\_handle must be returned by a RegisterGraph call

// to the same WorkerService.

string graph\_handle = 1;

// A unique ID to distinguish different runs of the same graph.

//

// The master generates a global unique `step\_id` to distinguish

// different runs of the graph computation. Subgraphs communicate

// (e.g., send/recv ops) with each other using `step\_id` to

// distinguish tensors generated by different runs.

int64 step\_id = 2;

// Options for this step.

ExecutorOpts exec\_opts = 5;

// Runs the graph.

//

// Sends the tensors in "send" into the graph before the run and

// fetches the keys into `RunGraphResponse.recv` after the run.

repeated NamedTensorProto send = 3;

repeated string recv\_key = 4;

// True if the RunGraphRequest is a partial run request.

bool is\_partial = 6;

// True if this is the last partial run request in a sequence of requests.

bool is\_last\_partial\_run = 7;

// If true then some errors, e.g., execution errors that have long

// error messages, may return an OK RunGraphResponse with the actual

// error saved in the status\_code/status\_error\_message fields of the

// response body. This is a workaround since the RPC subsystem may

// truncate long metadata messages.

bool store\_errors\_in\_response\_body = 9;

// Unique identifier for this request. Every RunGraphRequest must have a

// unique request\_id, and retried RunGraphRequests must have the same

// request\_id. If request\_id is zero, retry detection is disabled.

//

// Retried RunGraphRequests are problematic because they may issue a

// RecvTensor that will have no corresponding sender and will wait forever.

// Workers use request\_ids to reject retried RunGraph requests instead of

// waiting forever.

int64 request\_id = 11;

// Next: 12

}

message RunGraphResponse {

// A list of tensors corresponding to those requested by

// `RunGraphRequest.recv\_key`.

repeated NamedTensorProto recv = 1;

// If the request asked for execution stats, the cost graph, or the partition

// graphs, these are returned here.

// TODO(suharshs): Package these in a RunMetadata instead.

StepStats step\_stats = 2;

CostGraphDef cost\_graph = 3;

repeated GraphDef partition\_graph = 4;

// If store\_errors\_in\_response\_body is true in the request, then

// optionally the server may return an OK status for the RPC and

// fill the true status into the fields below, to allow for messages

// that are too long to fit in metadata.

error.Code status\_code = 5;

string status\_error\_message = 6;

}

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

//

// CleanupGraph method request/response messages

//

// After the master receives RunGraph responses from all workers, the

// master instructs every worker to cleanup any remaining state of a

// step (e.g. tensors buffered by a `Send` op but not picked up by

// other workers). The master does not necessarily need to wait for

// completion of CleanupGraph calls.

//

// Workers should cleanup step states automatically according to a

// TTL-based policy in case of master restarts.

//

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

message CleanupGraphRequest {

int64 step\_id = 1;

}

message CleanupGraphResponse {}

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

//

// RecvTensor method request/response messages

//

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

message RecvTensorRequest {

// The step in which the tensor will be produced.

//

// REQUIRED: This must eventually correspond to the `step\_id` passed

// into a RunGraph call on the same WorkerService.

int64 step\_id = 1;

// A key identifying the channel to receive tensors from. A RecvTensor request

// retrieves one tensor from the channel, but multiple tensors can be sent and

// received over the same channel with multiple RecvTensor requests. See

// rendezvous.h for details.

string rendezvous\_key = 2;

// If true, use an out-of-band DMA mechanism to transfer the

// received tensor.

bool dma\_ok = 3;

// Optional information on client-side device locality.

DeviceLocality client\_locality = 4;

// Optional information on server-side device locality.

DeviceLocality server\_locality = 5;

// Optional information needed by the RPC subsystem.

google.protobuf.Any transport\_options = 6;

// Unique identifier for this request. Every RecvTensorRequest must have a

// unique request\_id, and retried RecvTensorRequests must have the same

// request\_id. If request\_id is zero, retry detection and response cache

// are disabled.

//

// Retried RecvTensorRequests are problematic because a RecvTensor with no

// corresponding sender will wait forever, and the tensor may have been

// delivered to a previous retry. Workers use request\_ids to reject retried

// RecvTensor requests instead of waiting forever.

int64 request\_id = 7;

}

message RecvTensorResponse {

// The tensor as a proto.

TensorProto tensor = 1;

// If true, this tensor was the output of a dead node, and the

// content is invalid.

bool is\_dead = 2;

// The time at which tensor was available and started to be returned.

int64 send\_start\_micros = 3;

// Optional additional information about how to receive the tensor,

// e.g. in the event that `RecvTensorRequest.dma\_ok` was true.

google.protobuf.Any transport\_options = 4;

// Whether the receiver should send a MarkRecvFinishedRequest to the sender

// to ack the message.

bool require\_ack = 5;

}

// Message for managing the response cache maintained on the sender side.

// Currently only used by the gRPC worker service.

message MarkRecvFinishedRequest {

int64 request\_id = 1;

}

message MarkRecvFinishedResponse {}

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

//

// Logging method request/response messages

//

// NOTE(mrry): This feature is not supported in the open-source

// version, and these messages are expected to change.

//

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

// Out-of-band request to begin or end logging, or

// to retrieve logs for particular steps.

message LoggingRequest {

// If true, RPC logging will be enabled.

bool enable\_rpc\_logging = 1;

// If true, RPC logging will be disabled.

bool disable\_rpc\_logging = 4;

// If true, discard any saved logging data (for all steps).

bool clear = 2;

// When set, requests all saved log data pertaining to the step.

// Any log data retrieved is eliminated from the store and cannot be

// retrieved again.

repeated int64 fetch\_step\_id = 3;

}

message LabeledStepStats {

int64 step\_id = 1;

StepStats step\_stats = 2;

}

message LoggingResponse {

repeated LabeledStepStats step = 1;

}

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

//

// Tracing method request/response messages

//

// NOTE(mrry): This feature is not supported in the open-source

// version, and these messages are expected to change.

//

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

message TraceOpts {

// Length of the trace to be taken, in seconds.

double duration = 1;

// If true, capture step profile locally in each worker. Currently

// unimplemented.

bool use\_step\_profiler = 2;

// If true, capture kernel events from each worker.

bool use\_kernel\_profiler = 3;

// If true, capture extended profiling events from TensorFlow process.

bool use\_extended\_profiler = 4;

// If true, capture GPU profiling events locally on each

// machine. Currently unimplemented.

bool use\_gpu\_profiler = 5;

// If true, collect sampled profile events. Currently unimplemented.

bool use\_sample\_profiler = 6;

}

// Out-of-band request to configure distributed tracing.

message TracingRequest {

TraceOpts options = 1;

}

message TracingResponse {}

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

//

// Raw data transfers in support of Collective Ops.

// These methods are experimental and subject to change.

//

// The intention is to allow collectives to take advantage of the most

// efficient methods available on a platform, e.g. RDMA, and not be

// constrained to use the RPC system in use by other methods.

//

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

message RecvBufRequest {

// Use of the fields below may vary by implementation. For example

// the buf\_ptr and num\_bytes may be set only for local operations and

// not sent on the wire, or only sent on the wire in one direction.

// Used at server side to find the correct BufRendezvous.

int64 step\_id = 1;

// Arbitrary string identifying a BufRendezvous entry.

string buf\_rendezvous\_key = 2;

// Size of value expected, must agree with BufRendezvous entry.

int64 num\_bytes = 3;

// When RDMA is in use, address of destination field on client.

fixed64 buf\_ptr = 4;

// Optional information on client-side device locality.

DeviceLocality client\_locality = 5;

// Optional information on server-side device locality.

DeviceLocality server\_locality = 6;

// Optional, implementation-specific data.

google.protobuf.Any transport\_options = 7;

// For annotating timeline and device incarnation check.

string src\_device = 8;

// Optional, for annotating the timeline.

string dst\_device = 9;

// Depending on the RPC system in use, it may be necessary to set this

// id to detect resends of RPCs where the server is not aware that

// the prior RPC failed.

int64 request\_id = 10;

// Incarnation number of the source device, used to detect worker failures.

uint64 src\_incarnation = 11;

}

message RecvBufResponse {

// Use of the fields below may vary by implementation. Comments give

// intended use.

fixed64 buf\_ptr = 1; // Address of source field on server.

int64 num\_bytes = 2; // Byte length of buf\_ptr field, if set.

bool is\_dead = 3; // True if value is 'dead' like a tensor.

// Optional, implementation-specific data.

google.protobuf.Any transport\_options = 4;

// Optional, for timeline.

int64 send\_start\_micros = 5;

// Whether the receiver should send a MarkRecvFinishedRequest to the sender

// to ack the message.

bool require\_ack = 6;

}

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

//

// Collective Op dynamic group resolution messages.

//

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

// Supplies one or more device names as members of the group identified by

// group\_key. Service will respond when all group\_size devices become known.

// All devices in group must have same type.

message CompleteGroupRequest {

int32 group\_key = 1;

int32 group\_size = 2;

string device\_type = 3;

int32 collective\_type = 5;

DeviceAttributes device\_attributes = 6;

reserved 4;

}

// Gives the complete membership of the group identified by group\_key.

message CompleteGroupResponse {

int32 group\_key = 1;

int32 group\_size = 2;

string device\_type = 3;

int32 num\_tasks = 4; // number of distinct tasks hosting the devices

bytes communicator\_key = 7;

repeated DeviceAttributes device\_attributes = 8;

reserved 5, 6;

}

// Supplies data about one collective op belonging to the instance identified

// by instance\_key. Service will respond when all group\_size ops have

// become known. Most of the data being sent is for correctness checking,

// to ensure that all ops in the instance share common attributes.

message CompleteInstanceRequest {

string name = 1;

int32 type = 2;

DataType data\_type = 3;

TensorShapeProto shape = 4;

int32 group\_key = 5;

int32 group\_size = 6;

int32 instance\_key = 7;

string device\_type = 8;

repeated int32 subdiv\_offset = 9;

string device = 10;

bool is\_source = 11;

}

// Confirms that every op in the instance has consistently declared itself.

// Also gives the source\_rank in case of broadcast.

message CompleteInstanceResponse {

int32 instance\_key = 1;

int32 source\_rank = 2;

reserved 3;

}

// Request for next agreed-upon step\_id for the specified graph\_keys.

// This is used to enable multiple graphs containing nodes from

// a common collective instance to coordinate using the same step\_ids.

message GetStepSequenceRequest {

repeated int64 graph\_key = 1;

}

message StepSequence {

int64 graph\_key = 1;

int64 next\_step\_id = 2;

}

// Next valid step\_ids for one or more graph\_keys.

message GetStepSequenceResponse {

repeated StepSequence step\_sequence = 1;

}