syntax = "proto3";

package tensorflow;

import "tensorflow/compiler/xla/pjrt/distributed/protocol.proto";

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

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

// Represents a remote worker task, specified by job name and task id.

message CoordinatedTask {

string job\_name = 1;

int32 task\_id = 2;

}

// Status payload for all coordination service errors.

// Note: an empty proto may be set if the error is triggered by the task's own

// agent calls (i.e. not propagated by the service from another remote task).

message CoordinationServiceError {

// Removed fields which used to specify the error origin.

reserved 1, 2;

// If true, error is reported via the agent API by the user (and not an

// internal service error).

bool is\_reported\_error = 3;

// Denotes which task hit the error. If unset, the error originated from the

// same task that is processing this error.

CoordinatedTask source\_task = 4;

}

// Represent device information from different runtimes.

message TfDeviceList {

repeated DeviceAttributes devices = 1;

}

message XlaDeviceList {

xla.GlobalTopologyProto devices = 1;

}

message CoordinationServiceDeviceInfo {

oneof type {

TfDeviceList tf = 1;

XlaDeviceList xla = 2;

}

}

// Request and response messages for registering a worker to the cluster leader.

// Use `job` and `task` to represent the role of the worker, and use

// `incarnation` to uniquely identify a worker process. Leader responds with its

// `incarnation` to identify a leader process.

message RegisterWorkerRequest {

// Removed fields which used to specify the task.

reserved 1, 2;

fixed64 incarnation = 3;

// Moved the field `local\_device\_attributes` from this request message to

// WaitForAllTasksRequest defined below.

reserved 4;

CoordinatedTask source\_task = 5;

}

message RegisterWorkerResponse {

fixed64 leader\_incarnation = 1;

}

// Request and response messages for sending heartbeats.

message HeartbeatRequest {

// Removed fields which used to specify the remote task.

reserved 1, 2;

fixed64 incarnation = 3;

CoordinatedTask source\_task = 4;

}

message HeartbeatResponse {

fixed64 leader\_incarnation = 1;

// If there are failures in cluster, use additional metadata in response to

// broadcast error code and message to other workers.

}

// Request and response messages for waiting for all tasks.

message WaitForAllTasksRequest {

// Removed fields which used to specify the remote task.

reserved 1, 2;

// Removed field that specifically used TF device info.

reserved 3;

// All local device attributes on the request sender.

CoordinationServiceDeviceInfo local\_device\_info = 4;

CoordinatedTask source\_task = 5;

}

message WaitForAllTasksResponse {

fixed64 leader\_incarnation = 1;

// Removed field that specifically used TF device info.

reserved 2;

// All devices in the cluster.

CoordinationServiceDeviceInfo cluster\_device\_info = 3;

}

// Request and response messages for reporting errors to task.

message ReportErrorToAgentRequest {

int32 error\_code = 1;

string error\_message = 2;

// Removed fields that are embedded in payload.

reserved 3, 4;

CoordinationServiceError error\_payload = 5;

}

message ReportErrorToAgentResponse {}

// Request and response messages for reporting errors to service instance.

message ReportErrorToServiceRequest {

int32 error\_code = 1;

string error\_message = 2;

// Removed fields which used to specify the error origin.

reserved 3, 4;

CoordinatedTask error\_origin = 5;

}

message ReportErrorToServiceResponse {}

// Message for configuration key value.

// Key is structured like Unix file system, with multiple levels of directory

// names separated by the slash ('/') characters.

message KeyValueEntry {

string key = 1;

bytes value = 2;

}

// Request and response messages for inserting configuration key-value data.

message InsertKeyValueRequest {

KeyValueEntry kv = 1;

}

message InsertKeyValueResponse {}

// Request and response messages for getting configuration key-value data.

message GetKeyValueRequest {

string key = 1;

}

message GetKeyValueResponse {

KeyValueEntry kv = 1;

}

// Request and response messages for deleting configuration key-value data.

// When is\_directory is true, delete key-values recursively under `key`.

message DeleteKeyValueRequest {

string key = 1;

bool is\_directory = 2;

}

message DeleteKeyValueResponse {}

// Request and response messages for generic sync barriers.

message BarrierRequest {

string barrier\_id = 1;

int64 barrier\_timeout\_in\_ms = 2;

// Denotes list of tasks that will wait for the barrier. If unspecified, it

// implies that the entire cluster is participating in the barrier.

repeated CoordinatedTask tasks = 3;

// Task that is making the request.

CoordinatedTask source\_task = 4;

}

message BarrierResponse {}

// Request and response messages for cancelling generic sync barriers.

message CancelBarrierRequest {

string barrier\_id = 1;

// Task that is making the request.

CoordinatedTask source\_task = 2;

}

message CancelBarrierResponse {}

// Coordination Service defines a TensorFlow service that controls and

// coordinates distributed execution in a cluster of multiple workers.

//

// The service keeps track of the cluster configuration and the state of cluster

// members or the leader depending on the role of the current worker. The

// distributed runtime leverages this service to coordinate and perform cluster

// initialization, check the healthiness of workers, and propagate error

// messages to the cluster.

service CoordinationService {

// Register task to coordination service so that the service starts to track

// liveness of the task. RPC blocks and returns only when it registers to

// the service successfully, or error happens in the registering process.

rpc RegisterWorker(RegisterWorkerRequest) returns (RegisterWorkerResponse);

// Heartbeat message from task to coordination service. Heartbeat is sent from

// a task to refresh its timestamp on leader to avoid it becoming stale.

// RPC responds immediately after refreshing the timestamp on leader.

rpc Heartbeat(HeartbeatRequest) returns (HeartbeatResponse);

// Wait for all tasks in the cluster to be up and running. The RPC request

// only gets responded when all workers are registered, or some error occurs.

rpc WaitForAllTasks(WaitForAllTasksRequest) returns (WaitForAllTasksResponse);

// Report error to the task. RPC sets the receiving instance of coordination

// service agent to error state permanently.

// TODO(b/195990880): Consider splitting this into a different RPC service.

rpc ReportErrorToAgent(ReportErrorToAgentRequest)

returns (ReportErrorToAgentResponse);

// Report task error to coordination service. RPC sets the service-side task

// state to error, and propagate the error to other tasks in the cluster.

rpc ReportErrorToService(ReportErrorToServiceRequest)

returns (ReportErrorToServiceResponse);

// Insert configuration key-value that will be accessible to all cluster

// workers. The key can be formatted as Unix file path with hierarchy. The

// coordination service key-value store should only be used for cluster

// configuration data.

rpc InsertKeyValue(InsertKeyValueRequest) returns (InsertKeyValueResponse);

// Get configuration key-value. The request blocks until the key-value data

// becomes available (i.e., set by a worker in the cluster).

rpc GetKeyValue(GetKeyValueRequest) returns (GetKeyValueResponse);

// Delete configuration key-value. If is\_directory is set in request,

// recursively clean up all key-values under the path specified by `key`.

rpc DeleteKeyValue(DeleteKeyValueRequest) returns (DeleteKeyValueResponse);

// Blocks until all (or a subset of) tasks are at the barrier or the barrier

// fails.

//

// `barrier\_id` should be unique across barriers. Once the barrier has passed

// or failed, subsequent calls will not block, and immediately respond with

// the previous response.

//

// The first WaitAtBarrier() call received by the service for a particular

// barrier id is special in that it determines the barrier deadline based on

// timeout duration.

// However, if subsequent calls by different agents specify a different set of

// `tasks` for the same `barrier\_id`, the barrier will fail instantly.

//

// If no tasks are specified (default), the barrier will block for all the

// connected tasks.

//

// Possible service errors:

// - DeadlineExceeded: Timed out waiting for specified tasks at the barrier.

// Deadline is determined by the server timestamp when it receives the

// first WaitAtBarrier() + timeout duration.

// - Cancelled: One of the tasks called CancelBarrier().

// - Aborted: Service is shutting down.

// - Internal: Any participating task is in ERROR state.

// - InvalidArgument: (1) Conflicting tasks specified by different agents

// for the same barrier, (2) one of the participating tasks is not in

// the cluster, or (3) task making the request is not included in the

// list of participating tasks.

rpc Barrier(BarrierRequest) returns (BarrierResponse);

// Aborts the barrier if it is ongoing.

// Current and future WaitAtBarrier() calls with the same id will return a

// CANCELLED error status.

// Possible service errors:

// - FailedPrecondition: Barrier has already been passed.

// - NotFound: No barrier with the specified id is found.

rpc CancelBarrier(CancelBarrierRequest) returns (CancelBarrierResponse);

}