BEAR ROBOTICS

Bear Cloud API

Introduction

This document describes the gRPC based Bear Cloud API Service for third party clients.

Bear Cloud API service allows third parties to

1. send commands to control the robots,
2. receive status updates from robots (e.g. mission, battery status), and
3. get information robots.

For a detailed list of API capabilities, please refer to the Public Cloud API section below. Bear Cloud API uses the open source gRPC framework. In the future, corresponding REST APIs will be added.

Overview

Third party clients communicate with Bear Cloud API service via gRPC. For request / response type of communication, unary RPC is used. For communication where clients receive continuous updates from the Bear Cloud API service, server streaming RPC is used.

For server streaming RPCs, all responses have a metadata that contains a timestamp and a sequence number. The timestamp should not be used to order the responses since it is based on the local clock where the response was generated and not a global clock. Similarly, sequence number is guaranteed to be incremental which can be used to detect duplicate responses. The caveat is that the sequence number may be reset to 0 at any time but should be rare (only at restart of a service or robot). If a strict detection of response duplication is desired, both sequence number and timestamp should be utilized. Also note that streaming RPC queries will persist for a maximum of 60 minutes, if the query is still needed after this time.

When a gRPC call completes successfully the server returns an OK status along with the specified response to the client. If an error occurs, gPRC returns an error status code and string error message.

Message Encoding

gRPC uses Protocol Buffers which provide a serialization format for encoding and decoding, as well as Interface Definition Language.

Cloud API Service

Currently supported APIs are

```
syntax = "proto3";

package bearrobotics.api.v0.cloud;

service CloudAPIService {
  // List all robot IDs that satisfies the given filter options regardless
  // of robot status.
  rpc ListRobotIDs(ListRobotIDsRequest)
    returns (ListRobotIDsResponse) {}

  // Creates a mission to go charge a robot regardless of battery state.
  //
  // The call will fail if the robot is on a different mission.
  // Current mission needs to be canceled before the robot can be charged.
  // Mission can fail if no charger is available.
  rpc ChargeRobot(ChargeRobotRequest)
    returns (ChargeRobotResponse) {}

  // Creates a mission for a given type.
  //
  // Currently only supports ONEOFF which should have only one goal.
  // The call will fail if the robot cannot go on the requested mission.
  rpc CreateMission(CreateMissionRequest)
    returns (CreateMissionResponse) {}

  // Updates the specified mission with the given command.
  //
  // The call will fail if the robot is not on the specified mission
  // or cannot execute the command.
  // Currently only supports CANCEL.
  rpc UpdateMission(UpdateMissionRequest)
    returns (UpdateMissionResponse) {}
  // Subscribe to the robot mission status.
  //
  // Upon subscription, the latest known robot mission status is sent,
  // followed by updates whenever the mission status changes.
  rpc SubscribeMissionStatus(SubscribeMissionStatusRequest)
    returns (stream SubscribeMissionStatusResponse) {}

  // Subscribe to the robot battery status.
  //
  // Upon subscription, the latest known battery status is sent,
  // followed by updates whenever the battery status changes.
  rpc SubscribeBatteryStatus(SubscribeBatteryStatusRequest)
    returns (stream SubscribeBatteryStatusResponse) {}
}
```

Robot Information

List Robot IDs

```
rpc ListRobotIDs(ListRobotIDsRequest) returns (ListRobotIDsResponse) {}
```

Lists all robot IDs for a given location specified by the RobotFilter regardless of robot status. Note that if there are too many robots, not all robot IDs may be returned. Clients can compare total_robots and the number of robot_ids in the ListRobotIDsResponse to verify if all robot IDs have been returned.

```
syntax = "proto3";

package bearrobotics.api.v0.cloud;

message RobotFilter {
  // Empty location_id denotes all locations.
  string location_id = 1;
}

message ListRobotIDsRequest {
  RobotFilter filter = 1;
}

message ListRobotIDsResponse {
  int32 total_robots = 1;
  // This might not have all the robot IDs if there are too many.
  // It will have all the robot IDs if the number of robot_ids is same as
  // total_robots.
  repeated string robot_ids = 2;
}
```

To get a list of robot IDs from location_1 the request and response in ASCII proto format are

```
// Sample Request
filter {
  location_id: 'location_1'
}

// Sample Response
total_robots: 3
robot_id: 'robot_0'
robot_id: 'robot_1'
robot_id: 'robot_2'
```

Robot Commands

Create Mission

```
rpc CreateMission(CreateMissionRequest) returns (CreateMissionResponse) {}
```

Robots can be sent on an `ONEOFF` mission, i.e. a mission with only one `Goal`. Missions with multiple `Goal`s will be supported in the future. A `Goal` can either be a specific point in the map, referred as a `Destination`, or an area on the map such as a charging zone, referred as a `Zone`.

If the robot is currently on a different mission, the RPC will fail.

```
syntax = "proto3";

package bearrobotics.api.v0.robot;

// Point on the map.
message Destination {
  string destination_id = 1;
}

// Area on the map with one or more points where any one point is a valid
// goal for the robot.
message Zone {
  string zone_id = 1;
}

// Place on the map where the robot can be set to navigate towards.
message Goal {
  oneof goal {
    Destination destination = 1;
    Zone zone = 2;
  }
}

message Mission {
  enum Type {
    TYPE_UNKNOWN = 0;
    // Mission with only one goal.
    TYPE_ONEOFF = 1;
  }
  Type type = 1;

  // Currently only one goal is supported.
  repeated Goal goals = 2;
}

package bearrobotics.api.v0.cloud;

message CreateMissionRequest {
  string robot_id = 1;
  robot.Mission mission = 2;
}

message CreateMissionResponse {
  string mission_id = 1;
}
```

To send a `robot_5` to `Table_1` the request and response in ASCII proto format are

```
// Sample Request
```

```
robot_id: 'robot_5'
mission {
  type: TYPE_ONEOFF
  goals {
    destination: 'Table_1'
  }
}

// Sample Response
mission_id: '67aac595-d111-42e2-b910-d3a951228544'
```

Update Mission

```
rpc UpdateMission(UpdateMissionRequest) returns (UpdateMissionResponse) {}
```

If a robot is on a mission, the mission can be canceled by issuing a `CANCEL` command. If the robot is not on a mission the command is a `NOOP`.

If the robot is on a different mission than the specified mission the RPC will fail.

```
syntax = "proto3";

package bearrobotics.api.v0.robot;

// Action to update a current mission.
message MissionCommand {
  string mission_id = 1;

  enum Command {
    COMMAND_UNKNOWN = 0;
    COMMAND_CANCEL = 1;
  }

  Command command = 2;
}

package bearrobotics.api.v0.cloud;

message UpdateMissionRequest {
  string robot_id = 1
  robot.MissionCommand mission_command = 2;
}

message UpdateMissionResponse {}
```

To cancel the mission `67aac595-d111-42e2-b910-d3a951228544` for `robot_5` the request in ASCII proto format is

```
// Sample Request
robot_id: 'robot_5'
mission_command {
  mission_id: '67aac595-d111-42e2-b910-d3a951228544'
  command: COMMAND_CANCEL
}
```

Charge Robot

```
rpc ChargeRobot(ChargeRobotRequest) returns (ChargeRobotResponse) {}
```

Create a mission to go charge a robot. Similar to `CreateMission`, the RPC will fail if the robot is on a different mission. Note that the RPC can succeed, i.e. create a mission to charge the robot, but the mission might fail similarly to other missions. An example would be if a charger is not available.

```
syntax = "proto3";

package bearrobotics.api.v0.cloud;

message ChargeRobotRequest{
  string robot_id = 1;
}

message ChargeRobotResponse{
  string mission_id = 1;
}
```

To send a `robot_5` to go charge the request and response in ASCII proto format are

```
// Sample Request
robot_id: 'robot_5'
```

```
// Sample Response
mission_id: ‘211009052038826’
```

Robot Status Subscription

Mission Status

```
rpc SubscribeMissionStatus(SubscribeMissionStatusRequest)
  returns (stream SubscribeMissionStatusResponse) {}
```

Clients can subscribe to receive robot mission status updates. Upon subscription, the latest known robot mission status is received, followed by updates whenever there is a change in mission state.

```
syntax = "proto3";

package bearrobotics.api.v0.robot;

message MissionState {
  string mission_id = 1;

  enum State {
    STATE_UNKNOWN = 0;
    // Initial state when no mission has been run (e.g. empty feedback).
    STATE_DEFAULT = 1;
    STATE_RUNNING = 2;
    STATE_PAUSED = 3;
    STATE_CANCELED = 4;
    STATE_SUCCEEDED = 5;
    STATE_FAILED = 6;
  }
  State state = 2;

  // All goals for a given mission.
  repeated Goal goals = 3;
}

package bearrobotics.api.v0.common;

message EventMetadata {
  // The time when the event was recorded.
  google.proto.Timestamp timestamp = 1;

  // An incremental sequence number generated by the robot.
  // The sequence number should never be negative and can be reset to 0.
  // i.e. sequence is valid if it is larger than the previous number or 0.
  int64 sequence_number = 2;
}

package bearrobotics.api.v0.cloud;

message SubscribeMissionStatusRequest {
  string robot_id = 1;
}

message SubscribeMissionStatusResponse {
  common.EventMetadata metadata = 1;
  robot.MissionState mission_state = 2;
}
```

To receive mission status for `robot_7` the request and streaming responses in ASCII proto format are

```
// Sample Request
robot_id: 'robot_7'

// Sample Response
{
  metadata {
    timestamp {
      seconds: 1714312000
      nanos: 0
    }
    sequence_number: 75
  }
  mission_state {
    mission_id: 'mission_table_1'
    state: STATE_RUNNING
    goals {
      destination: 'Table_1'
    }
  }
}
```

```
{
  metadata {
    timestamp {
      Seconds: 1714315000
      nanos: 0
    }
    sequence_number: 76
  }
  mission_state {
    mission_id: 'mission_table_1'
    state: STATE_SUCCEEDED
    goals {
      destination: 'Table_1'
    }
  }
}
{
  metadata {
    timestamp {
      Seconds: 1714320000
      nanos: 0
    }
    sequence_number: 77
  }
  mission_state {
    mission_id: 'mission_table_2'
    state: STATE_RUNNING
    goals {
      destination: 'Table_2'
    }
  }
}
{
  metadata {
    timestamp {
      Seconds: 1714321000
      nanos: 0
    }
    sequence_number: 78
  }
  mission_state {
    mission_id: 'mission_table_2'
    state: STATE_CANCELED
    goals {
      destination: 'Table_2'
    }
  }
}
```

Battery Status

```
rpc SubscribeBatteryStatus(SubscribeBatteryStatusRequest)
  returns (stream SubscribeBatteryStatusResponse) {}
```

Clients can subscribe to receive battery status updates for a set of robots. Upon subscription, the latest known robot battery status is received, followed by updates whenever there is a change in battery state.

```
syntax = "proto3";

package bearrobotics.api.v0.robot;

message BatteryState {
  // State of charge percent from 0 (empty) ~ 100 (full).
  int32 charge_percent = 1;

  enum State {
    STATE_UNKNOWN = 0;
    STATE_CHARGING = 1;
    STATE_DISCHARGING = 2;
    STATE_FULL = 3;
  }
  State state = 2;
}

package bearrobotics.api.v0.common;

message EventMetadata {
  // The time when the event was recorded.
  google.proto.Timestamp timestamp = 1;

  // An incremental sequence number generated by the robot.
  // The sequence number should never be negative and can be reset to 0.
  // i.e. sequence is valid if it is larger than the previous number or 0.
  int64 sequence_number = 2;
}

package bearrobotics.api.v0.cloud;

// Filter for list of selected robots.
message RobotSelector {
  message RobotIDs {
    repeated string ids = 1;
  }
  oneof target_id {
    RobotIDs robot_ids = 1;
    // All robots for the location.
    string location_id = 2;
  }
}

message SubscribeBatteryStatusRequest {
  RobotSelector selector = 1;
}

message SubscribeBatteryStatusResponse {
  common.EventMetadata metadata = 1;
  string robot_id = 2;
  robot.BatteryState battery_state = 3;
}
```

To receive battery status for `robot_7` and `robot_5` the request and streaming responses in ASCII proto format are

```
// Sample Request
selector {
  robot_ids {
    ids: 'robot_7'
    ids: 'robot_5'
  }
}

// Sample Response
{
  metadata {
    timestamp {
      seconds: 1714312000
      nanos: 0
    }
    sequence_number: 100
  }
  robot_id: 'robot_5'
  battery_state {
    charge_percent: 95
    state: STATE_DISCHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714313000
      nanos: 0
    }
    sequence_number: 50
  }
  robot_id: 'robot_7'
  battery_state {
    charge_percent: 99
    state: STATE_CHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714314000
      nanos: 0
    }
    sequence_number: 101
  }
  robot_id: 'robot_5'
  battery_state {
    charge_percent: 95
    state: STATE_DISCHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714315000
      nanos: 0
    }
    sequence_number: 51
  }
  robot_id: 'robot_7'
  battery_state {
    charge_percent: 100
    state: STATE_CHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714316000
      nanos: 0
    }
    sequence_number: 52
  }
  robot_id: 'robot_7'
  battery_state {
    charge_percent: 100
    state: STATE_DISCHARGING
  }
}
```

To receive battery status for all robots at `location_1` which has `robot_0`,`robot_1`, and `robot_2` the request and streaming responses in ASCII proto format are

```
// Sample Request
selector {
  location_id: 'location_1'
}

// Sample Response
{
  metadata {
    timestamp {
      seconds: 1714322000
      nanos: 0
    }
    sequence_number: 10
  }
  robot_id: 'robot_0'
  battery_state {
    charge_percent: 95
    state: STATE_DISCHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714323000
      nanos: 0
    }
    sequence_number: 500
  }
  robot_id: 'robot_1'
  battery_state {
    charge_percent: 99
    state: STATE_CHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714324000
      nanos: 0
    }
    sequence_number: 1000
  }
  robot_id: 'robot_2'
  battery_state {
    charge_percent: 95
    state: STATE_DISCHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714325000
      nanos: 0
    }
    sequence_number: 1001
  }
  robot_id: 'robot_2'
  battery_state {
    charge_percent: 95
    state: STATE_DISCHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714326000
      nanos: 0
    }
    sequence_number: 501
  }
  robot_id: 'robot_1'
  battery_state {
    charge_percent: 100
    state: STATE_CHARGING
  }
}
{
  metadata {
    timestamp {
      seconds: 1714327000
      nanos: 0
    }
    sequence_number: 502
  }
  robot_id: 'robot_1'
  battery_state {
    charge_percent: 100
    state: STATE_DISCHARGING
  }
}
```

Authentication / Authorization

Clients are provided with API Key in JSON format which when supplied to the authentication server will return JWT. The JWT is the credential needed for authentication to the Bear Cloud API server and must be supplied with gRPC. The JWT expires after a set time which is specified by the `exp` field, so it is recommended to refresh the token regularly before expiry (every 15-20 minutes).

Credentials will be supplied in the format:

```
{
  "api_key": "deca1611-fb00-40b6-be4b-9ed797ae1642",
  "scope": "YOUR_SCOPE",
  "secret": "2cd66c59-a845-4f16-ac1e-50b50e3878ec"
}
```

All three fields must match in order for the credentials to be authorized.

- `api_key`: Uniquely identifies the credentials in our system.
- `scope`: Fixed at the time of API Key issue, and will be provided, it represents the distributor that the API Key is authorized to use.
- `secret`: Passcode of the `api_key`, it is important that this be stored securely.

The returned JWT should be attached to each outgoing gRPC query using a bearer authorization header with the following format: `"Authorization: Bearer <JWT>"`. Connections to the Bear Cloud API server are secured with TLS, the certificates for the API server are signed by Google which is a trusted Certificate Authority on many systems.

JWT Generation Example (with `curl`):

```
curl -X POST https://api-auth.bearrobotics.ai/authorizeApiAccess \
     -H "Content-Type: application/json" \
     -d '{"api_key":"deca1611-fb00-40b6-be4b-9ed797ae1642",
          "scope":"YOUR_SCOPE",
          "secret":"2cd66c59-a845-4f16-ac1e-50b50e3878ec"}'
```

Example client code using JWT for gRPC:

- Go with grpc-go:

```
type Token struct {
  JWT               atomic.Value
  TransportSecurity bool
}

// GetRequestMetadata gets the current gRPC request metadata,
// with current token. This func gets called before every gRPC query.
func (t *Token) GetRequestMetadata(_ context.Context, _ ...string)
(map[string]string, error) {
  if jwtToken := t.JWT.Load(); jwtToken != nil {
    return map[string]string{
      "Authorization": "Bearer " + jwtToken.(string),
    }, nil
  }
  return nil, fmt.Errorf("token is not set")
}

// RequireTransportSecurity indicates whether the credentials requires
// transport security.
func (t *Token) RequireTransportSecurity() bool {
  return t.TransportSecurity
}

func main() {
  token := Token{TransportSecurity: true}

  // Generate JWT from api-auth server using API Key.
  jwtToken := GetJWT(....)
  token.Token.Store(jwtToken)

  tls_creds := credentials.NewTLS(&tls.Config{})
  conn, err := grpc.Dial(serverAddress,
                         grpc.WithTransportCredentials(tls_creds),
                         grpc.WithPerRPCCredentials(token))

  // Make gRPC queries using conn

  // Update token.JWT when a new JWT is generated
}
```

- C++ Authentication Example
  - Replace `grpc::SSLCredentials(...)` call with `grpcServiceAccountJWTAccessCredentials(...)`
  - Header docs for JWT

- Python
  - Token-based authentication in gRPC
  - Adding a JWT as part of the request header

Changelog

- 2024-04-29: Initial version (ver 0.0).

- 2024-08-11: ver 0.0.1
    - Updated package path from `bearrobotics.api.[cloud|robot].v0` to `bearrobotics.api.v0.[cloud|robot]`.
    - Moved `EventMetadata` to package `bearrobotics.api.v0.common` from `bearrobotics.api.v0.[cloud|robot]`.
    - Changed `BatteryStatus.charge_percent` type from `float` to `int32`.
    - Updated Authentication / Authorization Section with API Auth server and JWT.
    - Streaming RPC connection will persist for a maximum of 60 minutes.