Interface Control Document
Urban Circuit

Revision 1
November 8, 2019

Defense Advanced Research Projects Agency
Tactical Technology Office
675 North Randolph Street
Arlington, VA 22203-2114

Table of Contents

1. Introduction

This document describes the interface to the DARPA Command Post server where teams competing in the DARPA Subterranean (SubT) Challenge will submit their artifact reports and mapping updates during the competition. The intent is to convey the overall concept of operations for interaction with the Command Post during competition and also to describe the hardware and software interfaces necessary to successfully interact with the server.

This document supersedes the *SubT Challenge Interface Control Document* dated February 11, 2019, finalized in advanced of the SubT Integration Exercise. Major revisions from the previous document are indicated by blue text. This document is subject to change and may be superseded by later versions. The latest official versions of all documents will be posted to the SubT Challenge Website and the SubT Community Forum.

2. Overview and Concept of Operations

The DARPA Command Post provides the sole interface by which teams will transmit competition data to DARPA for purposes of scoring, and, as such, is a critical element for competition participation. To facilitate ease of integration between each competitor and DARPA, the Command Post interface is designed to use widely accepted Internet standards, helping to ensure that teams will have plenty of prior experience building a compliant system and a wide variety of off-the-shelf libraries to choose from to help them do so.

The interface is utilized by transmitting data formatted in JavaScript Object Notation (JSON) or Concise Binary Object Representation (CBOR) using the Hypertext Transfer Protocol (HTTP) over a local Ethernet network between the competitor client and the DARPA Command Post server. The various functions of the interface are exposed as separate Uniform Resource Identifiers (URIs) that HTTP requests are executed against. Teams will authenticate their messages using an access token that will be provided during registration and setup.

There are two distinct types of interactions that teams will engage in with the DARPA Command Post: mandatory scoring interactions and optional (but strongly encouraged) mapping interactions. These distinct interaction types are described below. Furthermore, the two distinct types will be split into two distinct network endpoints, and more detailed specifications of the interfaces and the interaction protocols are described beginning in Section 3.

2.1. Scoring

During the competition, a team's score will be based solely on the Artifact Reports that the team submits to the DARPA Command Post via the scoring interface.

When a team is confident that they have identified and localized a new artifact, they will submit a POST request (a standard HTTP protocol operation) with a structure that describes the type and

location of the artifact to the appropriate URI for score reports. After receiving and processing the report, the server will return an HTTP response that contains a confirmation of the submitted report and an indication of its effect on the team's score. If there is a problem with the submitted request, for example an authentication token is not provided or the structure is malformed, a verbose error will be returned instead of an updated score. All properly formatted and authenticated artifact reports are recorded. Fields within the HTTP response body are used to indicate the effect of each report on the team's score (see POST /api/artifact_reports). For example, artifact reports made after the official run time has ended or in excess of the allotted report quantity have no effect on the score; this will be indicated by dedicated fields within the HTTP response body.

A separate URI will provide an interface for teams to request their current score, remaining artifact reports, and other information about the current run.

2.2. Mapping

Teams may provide a window into their competition progress by transmitting map, telemetry, and marker information from their system to the DARPA Command Post to aid in evaluation.

Mapping information consists of either two-dimensional or three-dimensional representations of the subterranean environment that the team has constructed using sensing and processing. Telemetry information consists of real-time positions for elements of the team's system. Marker information consists of any other visualization objects the team wishes to submit to DARPA, e.g., artifact detections, locations of dropped communications nodes, or traveled or planned trajectories of robots. Together, these types of information provide insight into the progress and success of a team's approach.

At desired intervals, the team will submit a POST request with a specified JSON or CBOR-formatted structure, encoding a standard Robot Operating System[1] (ROS) message, to an appropriate URI depending on the type of information. The server will return an HTTP response to indicate if the message was received correctly or if there was a formatting error. The message will be decoded into its corresponding ROS message and rebroadcast onto a running ROS network internal to the DARPA Command Post, where it will be consumed for visualization and analysis.

3. Physical and Network Interface

The DARPA Command Post physical and network interfaces are built on the traditional foundation of Ethernet communications, and so the general concepts should be familiar to all teams. We discuss here the specifics that teams will need to consider for successful interaction with the DARPA Command Post.

[1] http://www.ros.org

- ***Physical Interface:*** All network communications will take place over an Ethernet[2] connection carried by twisted-pair cabling, of sufficient bandwidth for scoring and mapping updates, between the DARPA Command Post and the team's base station.

  DARPA will provide one (1) clearly marked Ethernet cable with an RJ45 connector in the team base station area that will be the sole connection to the Command Post. Teams may connect this cable end into their system however they see fit, e.g., into a team-maintained network switch or directly into a computer network interface card.

- ***Network interface:*** Because there are two distinct types of interactions with the Command Post, i.e., both scoring and mapping information, these interactions will be handled by **two** (2) distinct network endpoints on DARPA-controlled servers. These endpoints will have static IP addresses, and teams will be responsible for ensuring that their specific network configuration can handle the required communication. Depending on the network configuration, this may require teams to, for example, bind multiple network addresses to their designated network interface in order to allow it to communicate on their internal network as well as with the DARPA Command Post.

  There will be one endpoint for scoring and one endpoint for mapping, and the static IP and port information for each endpoint will be printed and posted in the team base station area during setup. Teams should avoid using internal networking IP subnets and addresses that overlap/conflict with the following IP addresses nominally used by the DARPA network: 10.1.1.0/24, 10.2.1.0/24, 10.100.0.0/16. The addresses and ports will not change during a run; however, it is possible that DARPA may change the addresses and/or ports between runs.

  Network bandwidth for team interactions may be limited to approximately 100 Mbps over both network endpoints. Submitting data sizes near or over this limit may result in delays in score reports.

Note that communication with the Command Post is all in plain-text HTTP packets, and so network traffic between the team base station area and the DARPA Command Post will be separate from all other network traffic on-site to ensure security.

4. Protocol Interface

The DARPA Command Post protocols will be built on the common HyperText Transfer Protocol version 1.1[3] (HTTP/1.1), and so, the general concepts should be familiar to all teams. HTTP/1.1 utilizes the stream-oriented Transmission Control Protocol[4] (TCP) that handles the underlying mechanisms of forming and using socket connections. HTTP/1.1 allows requests to be targeted

[2] IEEE 802.3; twisted-pair cable, e.g., IEEE 803.3ab
[3] IETF RFC 7230 (https://tools.ietf.org/html/rfc7230)
[4] IETF RFC 1122 (https://tools.ietf.org/html/rfc1122)

at different Uniform Resource Identifiers[5] (URIs) that loosely describe the functionality that is being requested, and the Command Post uses different URIs to allow multiple functions to be provided at the same network endpoint. These URIs are described in detailed protocol descriptions of Sections 5 and 6.

Teams must include an authentication token with each HTTP packet that they send in order to act as an additional mechanism for ensuring that submissions were intended (particularly scoring submissions). The "Authorization" header field must include a 16-character "bearer" token[6] that will be provided for each team during site registration and setup. For example, if the team's token was "flux230{showroom", then they must include an "Authorization" header field with value "Bearer flux230{showroom". Please note that the DARPA Command Post will deviate from standards and will **not** provide a "WWW-Authenticate" response header with error information in the case of an authentication failure.

In an HTTP response, the HTTP status code will be used to indicate the success or failure of the request, and all HTTP response content will be in JavaScript Object Notation[7] (JSON), a self-describing format for representing structured data. Accordingly, the "Content-Type" header field will always be "application/json". Depending on the interface being used, this content could be either:

(1) a JSON-encoded string (i.e., in quotation marks) when there is a human-readable message accompanying this response (such as an error message)
(2) a JSON-encoded object (i.e., key-value pairs enclosed in braces) that contains detailed return information from an interaction and whose content will depend on the interaction

The applicable HTTP status codes are described below in Section 4.1.

Depending on the endpoint being used, HTTP request content may be in either JSON format or in Concise Binary Object Representation[8] (CBOR) format, a standardized binary serialization format loosely based on JSON. The "Content-Type" header field must be either "application/json" or "application/cbor" to indicate which format is being used. CBOR content may only be used on the mapping endpoint; for scoring, only JSON messages are allowed. The content that can be represented by CBOR is a superset of the content that can be represented by JSON, so translating between JSON and CBOR documents is straight-forward, and all specifications of request content will be presented in a JSON format for clarity. The CBOR content type is supported to enable more efficient transport of the binary data that is required for mapping interactions described in Section 6.

5 IETF RFC 3986 (https://tools.ietf.org/html/rfc3986), specifically the path information
6 IETF RFC 6750 (https://tools.ietf.org/html/rfc6750)
7 IETF RFC 8259 (https://tools.ietf.org/html/rfc8259)
8 IETF RFC 7049 (https://tools.ietf.org/html/rfc7049)

The allowable content types are summarized in the following table:

Endpoint	*HTTP* Request Type	*HTTP* Response Type
Scoring	JSON	JSON
Mapping	JSON or CBOR	JSON

4.1. HTTP Status Codes

The HTTP response status codes that will be returned by the Command Post, along with their meaning, are given below:

Code	*Description*	*Meaning*
200	OK	Request was accepted and/or response will be valid.
201	Created	Request was properly formatted and resource (i.e., artifact report) was created.
400	Bad request	JSON/CBOR parsing failed.
401	Unauthorized	Authentication token does not match expected token for this run (or was not provided).
404	Not Found	URI is not recognized
422	Unprocessable Entity	JSON/CBOR request had incorrect information for this interaction type.
429	Too many requests	Too many requests have been submitted in a given amount of time (limited to approximately 1 per second).

The associated response content for return code 200 will depend on the interaction, but the response content for all of the rest of the codes will be a human-readable error message encoded as a JSON string. For example, the following is a response for an “Unprocessable Entity” error:

```
HTTP/1.1 422 Unprocessable Entity
Content-Length: 23
Content-Type: application/json

“Missing field ‘type’”
```

5. Scoring Interaction Protocol

All scoring interactions will take place with the scoring endpoint as specified in Section 3 using the basic protocol described in Section 4. The specific URIs and JSON content related to scoring interactions are defined here. There are two functions implemented for the scoring interaction, and each is described in its own section below with the HTTP function and corresponding URI used as a section name.

The majority of regular interactions should take place using the “POST /api/artifact_reports” function (Section 5.2) at infrequent times. Teams should avoid overuse of the “GET /status” function (Section 5.1). API requests submitted at a rate exceeding approximately 1 per second may be throttled resulting in a HTTP 429 error response. Note: since the response to POST /api/artifact_reports includes updated status information, there is generally no need to immediately follow artifact report submissions with GET /api/status calls.

A test server that implements the scoring API has been posted to the SubT Challenge BitBucket repository: https://bitbucket.org/subtchallenge/test_scoring_server/src/master/. It can be used to verify scoring interaction submissions.

5.1. GET /api/status

Query the current run status, including time, score, and remaining reports.

The response content will be a JSON-encoded object with the following format:

```
{
    “score”: <integer>,
    “run_clock”: <float>,
    “remaining_reports”: <integer>,
    "current_team": <string>
}
```

The current score is an integer, the current clock time is given as seconds since the start of the run, and the remaining reports provides a count of how many score reports a team has remaining. Current team is a lowercase string representing the name of the team registered to the provided authentication token; it is provided as a confirmation.

Example request:

```
GET /api/status HTTP/1.1
Authorization: Bearer flux230{showroom
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 77
Content-Type: application/json

{“score”: 6, “clock”: 721.4, “remaining_reports”: 14, "current_team": "subt"}
```

5.2. POST /api/artifact_reports

Submit an artifact report and receive information about the score change.

The request content must be a JSON-encoded object with the following format:

```
{
```

```
  “x”: <float>,
  “y”: <float>,
  “z”: <float>,
  “type”: <string>
}
```

The “x”, “y”, and “z” values are the coordinates of the artifact location in the established DARPA reference frame (as defined in the Competition Rules document), and “type” is the string name of the artifact type as specified in the table below. Please see the Artifacts Specification Urban Circuit document for more information on the Urban Circuit artifacts. Note that the coordinates must be in meters and the type string must be spelled correctly (although case-insensitive). The reference frame will be the “darpa” frame, as introduced in Section 6.1. For additional information about the DARPA-defined reference frame, please see the Competition Rules document.

Urban Circuit Artifact Type Strings
Survivor
Backpack
Cell Phone
Vent
Gas

The response content will be a JSON object with the following format:

```
{
  "url": <string>,
  "id": <integer>,
  “x”: <float>,
  “y”: <float>,
  “z”: <float>,
  “type”: <string>

  "submitted_datetime": <string>,
  "run_clock": <float>,
  "team": <string>,
  "run": <string>,
  "report_status": <string>,
  “score_change”: <integer>
}
```

The semantics of the fields are as follows:

- *url* is a reference to the recorded artifact response following REST API convention.
- *id* is a unique identifier for the recorded artifact response.
- *x, y, z, and type* reflect the artifact report that was submitted, for reference

- *submitted_datetime* represents the absolute time of the report submission. It follows the ISO 8601 combined date and time format.
- *run_clock* is derived from *submitted_datetime* and corresponds to the number of seconds into the run of this report submission
- *team* represents the name of the team that submitted the report.
- *run* represents an identifier for the current run for this report.
- *report_status* indicates the status of the artifact report.
    - "scored" indicates the report was scored and one allotted artifact report was used.
    - "admin stop" indicates this report was not scored because it was submitted during an administrative stop.
    - "run not started" indicates this report was not scored because it was submitted prior to run start.
    - "report limit exceeded" indicates this report was not scored because there were no remaining reports available for this run.
    - "time limit exceeded" indicates that this report was not scored because it was submitted after the end of the run.
- *score_change* signifies the amount that the score changed as a result of this artifact report. For "scored" reports, its value will be either 0 (type and/or location were insufficient) or 1 (type and location were sufficient). For unscored reports, this field will always be 0.

Example request:

```
POST /api/artifact_reports HTTP/1.1
Authorization: Bearer flux230{showroom
Content-Length: 64
Content-Type: application/json

{“x”: 1011.242, “y”: -244.433, “z”: -10.011, “type”: “backpack”}
```

Example response:

```
HTTP/1.1 200 OK
Content-Length: 281
Content-Type: application/json

{"url": "http://<ipaddress>:<port>/api/artifact_reports/1", "id": 1, “x”: 1011.242, “y”: -244.433, “z”: -10.011,
    “type”: “backpack”, "submitted_datetime": "2019-01-01T00:00:00.000000-07:00", "run_clock": 721.4, "team":
    "subt", "run": 1, “report_status”: "scored", “score_change”: 1}
```

6. Mapping Interaction Protocol

All mapping, telemetry, and marker interactions will take place with the mapping endpoint as specified in Section 3 using the basic protocol described in Section 4. Map information is more complex than scoring information, but this complexity will be mitigated somewhat by reusing concepts and message formats that are part of the Robot Operating System (ROS) and thus widely used in the robotics community.

Mapping information will consist of periodic updates of the aggregate environment representation that teams have built using sensor data. Since one of the stated aims of the DARPA Subterranean Challenge is to develop technologies that provide rapid situational awareness, it is expected that teams will be maintaining some form of a central map at their base station. We will initially focus on two map representations that teams can provide to communicate their mapping progress: two-dimensional occupancy grids and three-dimensional point clouds. Other possible representations may be added, as necessary, depending on the needs of teams.

Telemetry information will consist of real-time positions of the resources within a team, for example the position and orientation of all of the robots, both ground and aerial, within the DARPA-established reference frame.

Marker information will consist of any other visualization objects the team wishes to submit to DARPA. This may contain artifact detections, locations of dropped communications nodes, traveled or planned trajectories of robots, mapping loop closure indicators, etc. Note that any artifact detections submitted through this marker update rather than the "POST /api/artifact_reports" function (Section 5.2) are unofficial and will not be scored.

Teams should avoid overuse of the "POST /map/update" function (Section 6.4); data submitted at a rate exceeding approximately 1 per second may be throttled resulting in a HTTP 429 error response.

A test server that implements the mapping API has been posted to the SubT Challenge BitBucket repository: https://bitbucket.org/subtchallenge/test_mapping_server/src/master/. It can be used to verify mapping interaction submissions.

6.1. Reference frames

The DARPA course reference frame, as defined in the Competition Guidelines, will be the reference frame used as a basis for all map information.

In common ROS usage, the reference frame for a message is often included in a "frame_id" field. A "frame_id" may be specified in the map information, but its value must always be "darpa" or an error will be returned (422 Unprocessable entity).

Because teams may be maintaining map information in a custom reference frame other than the standard "darpa" frame, map representations allow for the definition of a custom origin field that should interpreted as a rigid body transformation applied to the metric information within the message in order to place it into the "darpa" frame. This information will be under a field named "origin," and, if it is not present, it will be assumed to be identity and all metric information will be assumed to already be in the "darpa" frame.

6.2. Time stamps

Map data may also include timestamps to indicate when this particular information was generated or collected. It is not expected that teams will synchronize system clocks between the base station and the DARPA command post, and so teams should only ensure that all reported timestamps are monotonically increasing and internally consistent. These timestamps will be interpreted into the time reference of the DARPA command post as needed. If a timestamp is not specified, then the server will assume the time of receipt as the timestamp of the message.

6.3. Map formats

The DARPA Command Post will initially accept two different structures for map information, one for two-dimensional representations and one for three-dimensional representations. These structures correspond to ROS message definitions that are standard for their respective representations. The field names and values of these messages translate naturally into JSON elements except for the large byte arrays that make up the bulk of the message because JSON does not support binary data. This motivates the option of using the CBOR format for updates, but it is still possible to utilize JSON if desired. Teams must do one of the following:

(a) If the map update is being transmitted in JSON format, the byte array must be base64-encoded and sent as a string.

(b) If the map update is being transmitted in CBOR format, the byte array must be sent as a byte string (Major type 2) with no re-encoding.

Additionally, to further reduce the size of the data that must be transmitted, teams may compress the byte arrays before encoding the message. Only **gzip**[9] compression is supported at this time, but even basic gzip compression has proven to be extremely useful for occupancy grid data. DARPA will consider supporting other compression technologies if they are widely available and are demonstrated to be useful for supporting the aim of timely, rich map updates. If a team has compressed the map data byte array, they must include a "compression" field to indicate that compression is being used and signify the type of compression (i.e., "gzip").

Teams may choose to transmit partial map updates (i.e., a section of the aggregate map) to reduce the size of each message.

DARPA is open to accommodating additional map representations if they are not already covered by the provided ones, especially if they correspond with publicly available ROS message definitions and visualization tools (particularly *rviz* plugins).

[9] IETF RFC1952 (https://tools.ietf.org/html/rfc1952), reference implementation at https://zlib.net/

6.3.1. 2D Occupancy Grid

Two-dimensional occupancy grids are based off of the ROS *OccupancyGrid*[10] message in the *nav_msgs* package. This format describes a fixed-resolution grid of known dimensions where each cell is a byte that gives the integer probability that this cell is occupied, from 0 to 100%, or specifies that cell occupancy is unknown. An Occupancy Grid message is defined as:

```
{
  [optional] "header": {
    "stamp": <float>,
    "frame_id": <string>
  },
  "info": {
    "resolution": <float>,
    "width": <integer>,
    "height": <integer>,
    "origin": {
      "position": {
        "x": <float>,
        "y": <float>,
        "z": <float>
      },
      "orientation": {
        "x": <float>,
        "y": <float>,
        "z": <float>,
        "w": <float>
      }
    }
  },
  "data": <binary>,
    [optional] "compression": <string>
}
```

The semantics of the fields are as follows:

- *header* is an optional structure with timestamp and frame information that is intended to provide compatibility with standard ROS conventions, if desired. One or both of the subfields may be specified. If provided, *frame_id* **must** be "darpa" (see Section 6.1). If provided, *stamp* must be monotonically increasing and internally consistent (see Section 6.2).
- *resolution* is the size, in meters, of one grid cell (all cells are square)
- *width* is the number of grid cells in the +X direction of the map
- *height* is the number of grid cells in the +Y direction of the map

[10] http://docs.ros.org/api/nav_msgs/html/msg/OccupancyGrid.html

- *origin* gives the position and orientation (as a quaternion) of the map origin attached (attached to the lower-leftmost cell) in the “darpa” reference frame (see Section 6.1)
- *data* is an array of bytes, one for each cell in row-major order, of the integer occupancy probability from 0 to 100 or 255 to indicate that the cell is unknown; it must be an array that contains *width*height* bytes
- *compression* is an optional string to signify that the byte array has been compressed with the corresponding scheme; valid schemes are “none” or “gzip” and omitting the *compression* field indicates that compression is “none”. See Section 6.3.

Pictorially, the layout of the (uncompressed) data is:

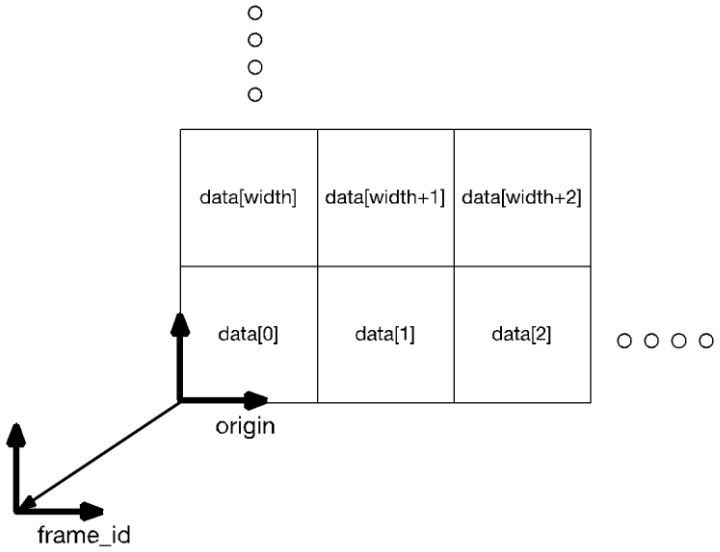

Figure 1: Uncompressed data layout for 2D Occupancy Grid

6.3.2.3D Point Cloud

Three-dimensional point cloud data is based off of the ROS *PointCloud2*[11] message from the *sensor_msgs* package. One feature of this message format is that each message also provides the schema that describes the layout of the binary data, and so the same message type can describe point clouds that provide additional scalar values, such as color channels or intensity, for each point in addition to 3D position.

A point schema consists of a collection of field schemas that describe each field belonging to that point. A single field schema is described by the JSON object:

<**field schema**> = {
  “name”: <**string**>,
  “offset”: <**integer**>,
  “datatype”: <**integer**>,

11 http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html

```
  "count": <integer>
}
```

The field semantics are:

- *name* is the description of this field, e.g., "x", "normal_y", or "rgba".
- *offset* specifies where this field begins (in bytes) from the beginning of the byte array describing this point
- *datatype* specifies the type for this field; it must be one of the following values:
    - 1 = 8-bit integer
    - 2 = 8-bit unsigned integer
    - 3 = 16-bit integer
    - 4 = 16-bit unsigned integer
    - 5 = 32-bit integer
    - 6 = 32-bit unsigned integer
    - 7 = 32-bit IEEE single-precision floating point
    - 8 = 64-bit IEEE double-precision floating point
- *count* is how many repeated copies of this datatype belong to this field (e.g., to describe a multi-element vector as one field)

A point schema is an array of field schemas:

```
<point schema> = [
  <field schema>
]
```

As an example, the following JSON object describes the schema for a point with x, y, and z coordinates as 32-bit floats and color as an RGBA value packed in a 32-bit unsigned integer:

```
[
  {"name": "x", "offset": 0, "datatype": 7, "count": 1},
  {"name": "y", "offset": 4, "datatype": 7, "count": 1},
  {"name": "z", "offset": 8, "datatype": 7, "count": 1},
  {"name": "rgba", "offset": 12, "datatype": 6, "count": 1}
]
```

A Point Cloud message is defined as:

```
{
  [optional] "header": {
    "stamp": <float>,
    "frame_id": <string>
  },
  [optional] "origin": {
    "position": {
```

```
      "x": <float>,
      "y": <float>,
      "z": <float>
    },
    "orientation": {
      "x": <float>,
      "y": <float>,
      "z": <float>,
      "w": <float>
    }
  }
  "fields": <point schema>,
  [optional] "is_bigendian": <bool>,
  "point_step": <integer>,
  "data": <binary>,
  [optional] "compression": <string>
}
```

The semantics are:

- *header* is an optional structure with timestamp and frame information that is intended to provide compatibility with standard ROS conventions, if desired. One or both of the subfields may be specified. If provided, *frame_id* **must** be "darpa" (see Section 6.1). If provided, *stamp* must be monotonically increasing and internally consistent (see Section 6.2).
- *origin* specifies a custom frame for the metric information in this point cloud. If provided, this field will be interpreted as a rigid-body transform that is applied to the spatial data in order to place it into the "darpa" frame (see Section 6.1). If omitted, the spatial data will be assumed to already be in the "darpa" frame.
- *fields* is an array of field schemas as described above
- *is_bigendian* is an optional Boolean that signifies that any multi-byte values are given in big-endian alignment; by default, all multi-byte values are assumed to be little-endian
- *point_step* specifies how big a single point is, in bytes (see note below)
- *data* is the binary point cloud data
- *compression* is an optional string to signify that the byte array has been compressed with the corresponding scheme; valid schemes are "none" or "gzip" and omitting the *compression* field indicates that compression is "none". See Section 6.3.

Though it may seem that the offsets and point step should be immediately calculable based on the datatypes involved, these are manually-specified to allow for padding in the structure of a point. For example, data structures on a 64-bit machine are often padded to be a size in bytes that is a multiple of 8 in order to allow the data to be more efficiently addressed and handled by the CPU.

Note that this Point Cloud message represents a subset of the ROS *PointCloud2* message definition with the addition of a custom origin (similar to a Point Cloud Library PCD file). In

particular, it is assumed that clouds will always be unstructured and that they will be “dense,f” meaning all points are valid.

6.4. POST /map/update

Send the latest version of the map to the Command Post.

The request must contain a JSON- or CBOR-encoded structure as follows:

```
{
  “type”: <string>,
  “msg”: <map>
}
```

The *type* field describes which of the two map formats is being sent, and the *msg* field contains the actual map data. The values are as follows:

Map type	*“type”*	*“msg”*
Occupancy grid	“OccupancyGrid”	See Section 6.3.1
Point cloud	“PointCloud2”	See Section 6.3.2

The response will be empty (JSON null), but the HTTP status code will indicate if the message was accepted (i.e., code 200).

6.5. POST /state/update

Send the latest telemetry information to the Command Post.

The request must contain a JSON- or CBOR-encoded structure as follows:

```
{
  [optional] “header”: {
    “stamp”: <float>,
    “frame_id”: <string>
  },
  “poses”: [
    {
      [optional] “name”: <string>,
      “position”: {
        “x”: <float>,
        “y”: <float>,
        “z”: <float>
      },
      “orientation”: {
        “x”: <float>,
        “y”: <float>,
        “z”: <float>,
        “w”: <float>
      }
    },    [...]
  ]
}
```

The field semantics are:

- *header* is an optional structure with timestamp and frame information that is intended to provide compatibility with standard ROS conventions, if desired. One or both of the subfields may be specified. If provided, *frame_id* **must** be “darpa” (see Section 6.1). If provided, *stamp* must be monotonically increasing and internally consistent (see Section 6.2).
- *poses* is an array of similarly structured objects that contains all of the three-dimensional poses (position and quaternion orientation) of the elements of the system, defined in the “darpa” reference frame. Each pose may optionally include a *name* for the element, which should represent the robot name and match the identification reported to DARPA on the robot roster at registration and check-in.

This message is essentially the ROS *PoseArray*[12] from the *geometry_msgs* package with the addition of a custom name field that can be ignored.

The response will be empty (JSON null), but the HTTP status code will indicate if the message was accepted (i.e., code 200).

6.6. POST /markers/update

Send the latest marker information to the Command Post.

The request must contain a JSON- or CBOR-encoded structure as follows:

```
{
  “markers”: [
    {
      [optional] “header”: {
        “stamp”: <float>,
        “frame_id”: <string>
      },
      [optional] “ns”: <string>,
      [optional] “id”: <int>,
      “type”: <int>,
      “action”: <int>,
      “pose”: {
        “position”: {
          “x”: <float>,
          “y”: <float>,
          “z”: <float>
        },
        “orientation”: {
          “x”: <float>,
          “y”: <float>,
```

12 http://docs.ros.org/api/geometry_msgs/html/msg/PoseArray.html

```
            "z": <float>,
            "w": <float>
          }
        },
        "scale": {
          "x": <float>,
          "y": <float>,
          "z": <float>
        },
        "color": {
          "r": <float>
          "g": <float>
          "b": <float>
          "a": <float>
        },
        [optional] "lifetime": <duration>
       [optional] "text": <string>
    },    [...]
  ]
}
```

The field semantics for each element of the *markers* array are listed here and in the documentation for ROS *MarkerArray*[13] from the *visualization_msgs* package:

- *header* is an optional structure with timestamp and frame information that is intended to provide compatibility with standard ROS conventions, if desired. One or both of the subfields may be specified. If provided, *frame_id* **must** be "darpa" (see Section 6.1). If provided, *stamp* must be monotonically increasing and internally consistent (see Section 6.2).
- *ns* is an optional namespace identifier for the object
- *id* is an optional number identifier for the object
- *type* is an indicator of the marker type (e.g., Arrow, Sphere), which is further documented in the ROS *MarkerArray* message documentation
- *action* represents one of the following: 0 = add/modify the marker, 2 = delete the marker, 3 = delete all markers
- *pose* is the three-dimensional pose (position and quaternion orientation) of the object, defined in the "darpa" reference frame
- *scale* is the three-dimensional size of the object in meters
- *color* is the specified display color of the object, specified in RGB-A format with values between 0 and 1. The A "alpha" value must be set to a nonzero value, or the object will appear invisible.
- *lifetime* is an optional duration value to indicate the period of time after which the marker should be automatically deleted. The default value is 0, which displays the marker indefinitely.

[13] http://docs.ros.org/api/visualization_msgs/html/msg/MarkerArray.html

- *text* is the label to display if the object is a text marker type

The response will be empty (JSON null), but the HTTP status code will indicate if the message was accepted (i.e., code 200).

Appendix A – Standards/References

The following are the applicable standards and references for this document:

- Ethernet: IEEE 802.3; twisted-pair cable, e.g., IEEE 803.3ab
- HyperText Transfer Protocol 1.1: IETF RFC 7230 (https://tools.ietf.org/html/rfc7230)
- TCP/IP: IETF RFC 1122 (https://tools.ietf.org/html/rfc1122)
- URI: IETF RFC 3986 (https://tools.ietf.org/html/rfc3986), specifically the path information
- Bearer Token Authentication: IETF RFC 6750 (https://tools.ietf.org/html/rfc6750)
- JSON: IETF RFC 8259 (https://tools.ietf.org/html/rfc8259)
- CBOR: IETF RFC 7049 (https://tools.ietf.org/html/rfc7049)
- gzip compression: IETF RFC1952 (https://tools.ietf.org/html/rfc1952), reference implementation at https://zlib.net
- ROS: http://www.ros.org
- ROS OccupancyGrid message: http://docs.ros.org/api/nav_msgs/html/msg/OccupancyGrid.html
- ROS PointCloud2 message: http://docs.ros.org/api/sensor_msgs/html/msg/PointCloud2.html
- ROS PoseArray message: http://docs.ros.org/api/geometry_msgs/html/msg/PoseArray.html
- ROS MarkerArray message: http://docs.ros.org/api/visualization_msgs/html/msg/MarkerArray.html

Appendix B – Command Post Architecture

The functionality of the Command Post can be roughly represented with the following diagram:

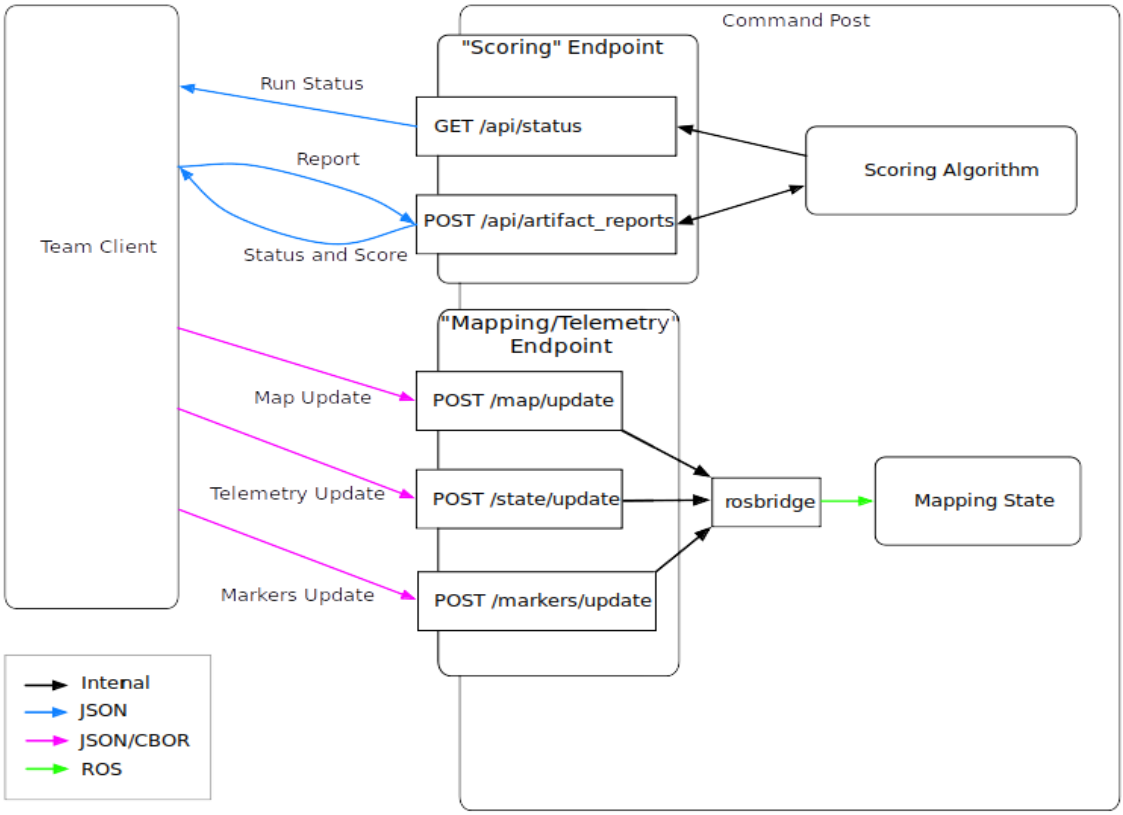

Figure 2: Command Post Architecture Diagram

Note that, internally, the Command Post will use ROS as a transport and application framework for using mapping information provided by teams.