The Navigation Command Protocol Application Programming Interface

The **Professor Aardvark's Asteroid Journey** (PAAJ) **Formation Challenge** (FC) **Navigation Command Protocol** (NCP) **Application Programming Interface** (API) is the API used to send ZeroMQ network requests from the player's navigation application to their FC client, as well as the description of the JSON response objects returned.

The return values that contain ship information identify the ship by a ship ID. The ship ID shall remain constant throughout an individual challenge. One ID will identify the same ship across all command responses within a challenge, but the ID for that ship may be different during another challenge.

The Commands

The available commands via the NCP API are listed below. In the sections that follow each command is described in more detail with response object definitions as well as sample code for usage of each command for each language of the provided messaging classes.

Command	Action
configuration	return configuration parameters for challenge
locate	return sensor scan of drone IDs, positions and velocities
health	return a drone ID and health value for each drone ship
thrust +x	pulse thruster to accelerate in the +x direction
thrust +y	pulse thruster to accelerate in the +y direction
thrust -x	pulse thruster to accelerate in the -x direction
thrust -y	pulse thruster to accelerate in the -y direction
status	return score and challenge time information

Create NCP API Instance

For each language a sample of how to send a command is provided below. In order to use the AardMsg class in your code you just follow the installation instructions provided with that language package for the particular operating system your team is using for development and competition. These are provided in each language distribution download.

Python

```
# Import the API definition
from aardmsg import AardMsg

# Create an API instance, and connect to the app
am = AardMsg(host='localhost')

# Send a message (request the game configuration)
msg = 'configuration'
am.send(msg) # TODO returns true if worked
print('Sent: ', msg)

# Wait up to the provided time in ms to receive the response
wait_ms = 100
rsp = am.recv(wait_ms)
```

```
print('Received: ', rsp)

# when finished close the connection
am.disconnet()
```

Java

```
// Create an API instance, and connect to the app
AardMsg am = new AardMsg("localhost");
am.connect(); // TODO workign to remove this

// Send a message (request the game configuration)
std::string msg = "configuration";
am.send(msg); // TODO returns true if worked
System.out.println("Sent: " + msg);

// Wait up to the provided time in ms to receive the response
int wait_ms = 100;
std::string rsp = am.recv(wait_ms);
System.out.println("Received: " + rsp);

// when finished close the connection
am.disconnect();
```

C++

```
// Create an API instance, and connect to the app
AardMsg am = new AardMsg("localhost");

// Send a message (request the game configuration)
std::string msg = "configuration";
am.send(msg); // returns true if worked
std::cout << "Sent: " << msg << std:endl;

// Wait up to the provided time in ms to receive the response
int wait_ms = 100;
std::string rsp = am.receive(wait_ms);
std::cout << "Received: " << rsp << std:endl;

// when finished close the connection
am.disconnect();
```

C

Note the C# interface is not yet provided, but is in development to provide the following:

```
// Create an API instance, and connect to the app
AardMsg am = new AardMsg("localhost");

// Send a message (request the game configuration)
string msg = "configuration";
am.send(msg); // returns true if worked
Console.WriteLine("Sent: " + msg);
```

```
// Receive the response
// Wait up to the provided time in ms to receive the response
int wait_ms = 100;
string rsp = am.receive(wait_ms);
Console.WriteLine("Received: " + rsp);

// when finished close the connection
am.disconnect();
```

Command: configuration

The `configuration` command queries the game for the challenge parameters. These values may vary from challenge to challenge, but will remain constant for a given run. So it should only be necessary to send the configuration query once per run.

These are set at the start of the run, and shouldn't change for the duration, so students should only need to get them once.

Property	Type	Notes
version	string	server version in case the server API changes
formation_radius	float	the radius of the formation circle around the mother ship in meters
distance_tolerance	float	the maximum allowable radial deviation in meters from the formation circle boundary
angular_tolerance	float	the maximum allowable angular deviation from equiangular spacing around the formation circle
mothership_aard_field_radius	float	the radius of the circular forcefield boundary around the mothership
droneship_aard_field_radius	float	the radius of the circular forcefield boundary around each drone
time_limit	float	given as starting value in minutes, not the current time left
velocity_increment	float	the velocity increase in meters/second for each thruster pulse
drone_count	int	the number of drone ships in the challenge
drone_id	int	unique ID of your drone ship

Example JSON response

```
{
    "version": "1.0.0",
    "formation_radius": 40.0,
    "distance_tolerance": 10.0,
    "angular_tolerance": 5.0,
    "mothership_aard_field_radius": 20.0,
    "droneship_aard_field_radius": 10.0,
    "time_limit_seconds": 300.0,
    "velocity_increment": 1.0,
    "drone_count": 1,
    "drone_id": 0
}
```

Command: locate

The `locate` command returns information relevant to controlling the ship: the ID, the position, and the velocity for each drone ship (including the local player). The position is a 2D point in meters with the mother ship in the center at (0, 0). The velocity is a 2D vector in meters/second. This API is likely to be hit a lot.

Property	Type	Notes
version	string	server version
ships	array	array of ship positions and velocities
ships.id	int	ship ID
ships.position	array	the ship's position in meters, relative to the mothership
ships.position.x	float	the ship's position X coordinate
ships.position.y	float	the ship's position Y coordinate
ships.velocity	array	the ship's velocity in meters per second, relative to the mothership
ships.velocity.x	float	the ship's velocity X coordinate
ships.velocity.y	float	the ship's velocity X coordinate

Example JSON response


```
{
    "version": "1.0.0",
    "ships": [
        {
            "id": 0,
            "position_x": -19.799999237060548,
            "position_y": 30.100000381469728,
            "velocity_x": 0.0,
            "velocity_y": 0.0
        },
        {
            "id": 1,
            "position_x": 20.0,
            "position_y": -15.0,
            "velocity_x": 0.0,
            "velocity_y": 0.0
        }
    ]
}
```


Command: health

The `health` command returns a drone ID and a health value for each drone ship. The drone IDs match the drone IDs returned by the `locate` command. The health value is a percentage, where 100% is perfect health, and 0% is dead.

Any other conditions of the other ships would also go here - this one is kind of intended as the "teamwork encourager" API.

Property	Type	Notes
version	string	server version
ships	array	array of ship health information
ships.id	int	ship ID
ships.health	int	current health value for ship
ships.health_change_rate	float	instantaneous health rate of change?

Example JSON response

```
{
    "version": "1.0.0",
    "ships": [
        {
            "id": 0,
            "health": 100.0,
            "health_change_rate": 0.0
        }
    ]
}
```


Command: thrust

The `thrust` command is the only command that includes an argument of either +x, -x, +y or -y to specify the direction of the thrust. The return value is similar to that of locate, but only for the particular ship just commanded.

Property	Type	Notes
id	int	ship ID
position_x	float	the ship's position X coordinate
position_y	float	the ship's position Y coordinate
velocity_x	float	the ship's velocity X coordinate
velocity_y	float	the ship's velocity X coordinate

Example JSON response


```
{
    "id": 0,
    "position_x": -19.799999237060548,
    "position_y": 30.100000381469728,
    "velocity_x": 2.0,
    "velocity_y": 0.0
}
```


Command: status

The `status` command returns a scan of stuff going on in the game. This information is likely fairly irrelevant, but some teams might find some use for it. They've proven they're capable of some pretty brilliantly weird things in the past.

Property	Type	Notes
version	string	server version
remaining_challenge_time	float	the number of seconds left in the challenge
glow_active	bool	true if the glow state is active
ships	array	array of ship status information
ships.id	int	ship ID
ships.glow_score	int	score accumulated from glow (or should it be glow duration?)
ships.total_score	int	the total game score (so far) for the ship

Example JSON response

```
{
    "version": "1.0.0",
    "remaining_challenge_time": 251.62803649902345,
    "glow_active": true,
    "ships": [
        {
            "id": 0,
            "glow_score": 48.371971130371097,
            "total_score": 0.0
        }
    ]
}
```


Network Configuration Options

There are several network configuration options available for the NCP API usage. In most cases the simplest to be used will match one of the following where the provided formation client and navigation application run on the same machine either locally or in the cloud as shown below:

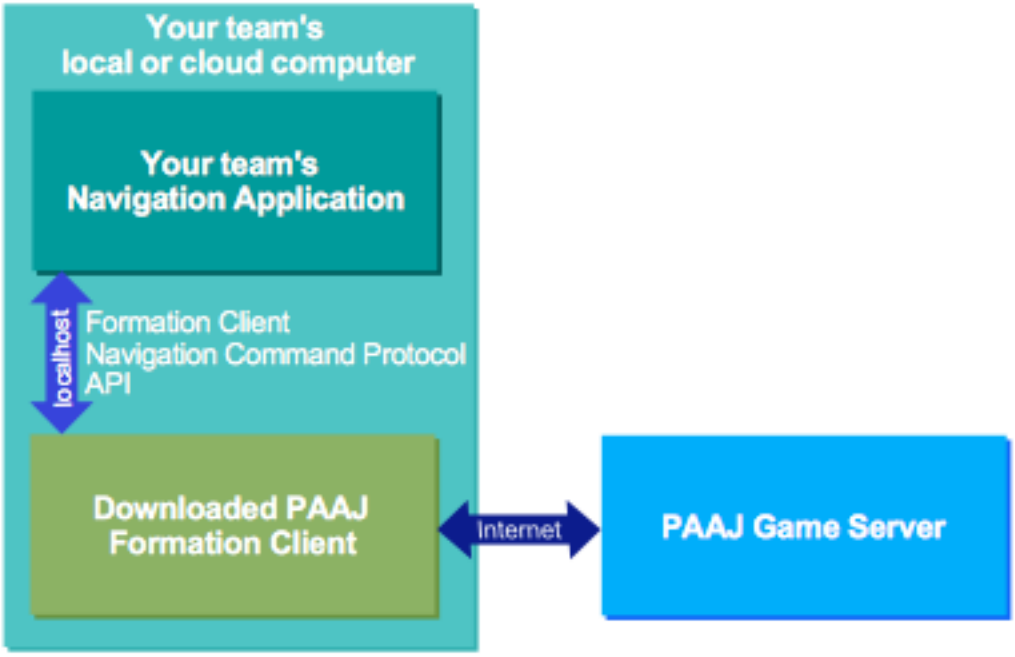

Figure 1: Image of standard network configuration

If running from the cloud the screen data must be coming from the cloud. At this time we are working to allow navigation applications not on the same device and possibly navigation applications from the cloud to access local machines during the competition, however, the security configuration to allow that is still in work. As such there is some chance these configurations will not be supported, so please be sure to bring a solution to game day that supports your navigation application running locally with the formation client instance you use for competition.