ACS - Mission Configuration Server

This document describes an automated boot-time capability for updating autopilot configurations from a central server.

The challenge

The currently-used autopilot is the Pixhawk running Ardupilot firmware. The firmware defines four types of configurable entities: parameters, way points, fence points, and rally points. Ground control software such as MAVProxy provides facilities for loading and saving these configurations as files.

Although the payload computer does not currently hold mission-specific configurations, it may also do so in the future.

As the number of aircraft in a fleet increases, so does the challenge of ensuring that all aircraft have the correct configuration loaded prior to each flight. Note that configurations may be specific to certain aircraft or to certain missions. A mechanism is needed to ensure that correct configurations are loaded.

Stacks and slots

The current concept of operation for this swarm entails the use of "stacks" (lateral points where aircraft gather once airborne and prior to executing swarm behaviors) and "slots" (altitude offsets within a stack). As an example, a 50-plane mission might be executed by defining two stacks, each with 25 slots. Each plane is assigned to a unique (stack, slot) prior to takeoff, thereby deconflicting airspace between aircraft in the absence of any active collision avoidance measures. See the discussion below about templates, and launch.md, for more details.

Fetching and loading configurations

The payload software includes task_runner.py in the ap_tasks package. It is conceptually similar to a Linux init system in that it triggers various tasks to execute depending on the state of the aircraft. For example, tasks may be run when the aircraft software first starts up, or when the throttle is armed.

Two task classes, FetchConfigTask and VerifyConfigTask, are defined for fetching configurations from a server and loading those configurations onto an autopilot, respectively.

FetchConfigTask
The fetching task waits until the payload software is fully loaded and communications with the autopilot are established (to ensure that configurations can be loaded onto the autopilot) before proceeding. It then attempts to download several files from a configuration server (currently hard-coded to 192.168.2.1, the default gateway of team 1):

http://192.168.2.1/XXX/fence
http://192.168.2.1/XXX/param
http://192.168.2.1/XXX/rally
http://192.168.2.1/XXX/rally.template
http://192.168.2.1/XXX/wp
http://192.168.2.1/XXX/wp.template

In the URLs above, XXX is the 0-padded ID of the aircraft (e.g., aircraft 13 would attempt to fetch http://192.168.2.1/013/fence and so on). This way, it is possible to support aircraft-specific configurations.

All files except those with the .template suffix are structured in file formats used by MAVProxy (at the time of this writing) for their respective configuration types. The two template files are discussed further below.

All fetched files are stored in ~/blessed/ (the files are referred to as "blessed" configurations). Prior to the first attempted fetch, any blessed files from previous flights are deleted. If any files cannot be fetched, the corresponding files in ~/blessed/ will be absent or empty.

**VerifyConfigTask**

Once all fetching is complete, the configuration loading ("verification") task runs (see pyload.md for details on task ordering). This task iterates over the four non-template files, loading each into the autopilot by calling the /autopilot/load_* ROS services provided by the file module in autopilot_bridge, where * is expanded to each configuration type.

For rally, fence, and way points, the loaded file will overwrite the existing autopilot configuration. For parameters, all parameters in the file will overwrite corresponding autopilot parameters; parameters not listed in the file will be untouched. This is important as each aircraft maintains certain tuning parameters that should not be overwritten.

Before loading any configurations, the task sets ROS parameters ok_* to false for each configuration type. The load services report back a true or false status for success of the load; the task sets the "ok" flag accordingly once the service completes. These flags are incorporated into the FlightStatus message that is sent periodically to the ground, and is used by ground operators to determine whether configurations have been successfully loaded.

Finally, the task commands the autopilot to reload waypoint 1 into memory so that the autopilot is using the waypoint loaded from file.
Configuration server

As noted above, the fetching task is hard-coded to fetch configuration files from a web server at 192.168.2.1. For team 2, this value can be changed in code (or a ROS parameter could be introduced to set it). Alternatively, both teams' configurations can be managed on a single server that is network-accessible from both networks (this is the configuration that has been field-tested).

The only requirement for the web server is to make the configuration files available at the URLs described above. As an example, the configuration that has been used in field testing is provided here, using the Apache web server (version 2.x) on Ubuntu Linux.

Configure the system to have the IP address 192.168.2.1 on the interface that connects to the swarm network (see wireless.md for configuration details).

After installing the web server (sudo apt-get install apache2), make sure that the Apache configuration file /etc/apache2/apache2.conf is updated so that the following Directory entry looks as follows:

```xml
<Directory />
  Options Indexes FollowSymLinks
  AllowOverride None
  Require all granted
</Directory>
```

Also ensure that the file /etc/apache2/sites-enabled/000-default.conf exists and contains the following line:

```plaintext
DocumentRoot /var/www/html
```

Then create the following directory structure:

```
/var/
  www/
    html/
      001/
      002/
      ... for all aircraft IDs in use ...
    templates/
```

Note that http://192.168.2.1/123/fence will point to the file /var/www/html/123/fence.

The simplest and most general way to store configurations is to copy all "blessed" (human-approved) files (fence, param, rally, and wp) into each numbered directory. However, it is likely that many or all aircraft will share some configuration. In this case, copy all configuration files into the templates directory. Then create soft links (file shortcuts) in one numbered directory, and copy its contents into the other numbered directories:

```bash
for t in /var/www/html/templates/*; do ln -s $t /var/www/html/001/; done
cd /var/www/html
for f in [0-9]*; do cp 001/* $f; done
# Note that the copy from 001/ to 001/ will give an error; ignore this
```
Now, any adjustments to the blessed files only need to be made once, in templates. If any aircraft need their own specialized configurations, simply delete the soft links from their numbered directories, and place their configurations in those directories instead.

### Mission templates

While the four configuration files fence, param, rally, and wp are used to statically configure an aircraft's geo-fence points, parameters, rally and way points, respectively, the files rally.template and wp.template are treated specially. They are downloaded during the fetch task, but are ignored during the verification task.

These two "template" files are used to support the "stack and slot" configuration discussed above. They are in the format of regular rally and way point files, but with special syntax added so that, with certain inputs applied, the files can be transformed into rally and way point files for a specific stack and slot.

As an example, consider the rally.template file below:

```
STACK_1 RALLY 35.719882 -120.768242 STDALT.000000 40.000000 0.000000 0
STACK_2 RALLY 35.720323 -120.769559 STDALT.000000 40.000000 0.000000 0
```

As described in launch.md, a stack and slot are chosen prior to takeoff. When those inputs are sent to the aircraft, function net_mission_config in network.py runs. It processes the template files (if present) with these inputs, and generates an output file. Lines that start with STACK_# (where # is the chosen stack number) and lines that do not start with STACK_ are included in the output file. All instances of STDALT are replaced with the chosen slot altitude. The resulting output is in the format of a normal rally file, which is then loaded into the autopilot. The same process occurs for the waypoint file.

Function net_mission_config also sets parameters based on the chosen slot altitude, and updates the ok_* ROS parameters accordingly.

A common use case when using templates is to place only the fence, param, rally.template, and wp.template files in numbered folders on the configuration server. After the aircraft automatically fetches and verifies just the fence and param files, the ok_ral and ok_wp parameters will still be set to false. When the stack and slot configuration is sent to the aircraft, the ok_prm parameter will be set to false. Then, after the new rally and wp files are generated and loaded, and the updated altitude parameters are set, all ok_* parameters will be set to true.

Template files can be generated using update_templates.py in acs-env. Review of this code is strongly recommended before using its results; it takes standard input files in ~/ACS/acs-env/data/missions/swarming/ and produces template files with the syntax additions described above.