ACS - Aircraft Payload Software

This document describes the **autonomy payload**: the secondary or companion computer and software that wraps the aircraft autopilot and implements swarm capabilities, including air-to-air and air-to-ground communications, swarm state tracking, and a framework for implementing swarm behaviors, also known as controllers.

Foundations

The autonomy payload is a collection of Robot Operating System (ROS) packages, along with libraries, supporting scripts, and ROS launch files, that are designed to run on a secondary or companion computer running Linux onboard each plane. This companion computer is intended to interface with the plane's autopilot, any onboard sensors (e.g., cameras), and a wireless network that connects planes with each other and ground stations.

Nearly all of the current payload software is written in Python (v2.7), and is tested with ROS Indigo on Ubuntu 14.04 LTS.

File structure

The file structure of installed payload is shown below; directories containing payload nodes are denoted with an *.

```
acs_ros_ws/
  src/
    autopilot_bridge/ *
    autonomy-payload/
      ap_lib/
      ap_logging/ *
      ap_master/
      ap_mission_planning/ *
      ap_msgs/
      ap_network_bridge/ *
      ap_path_planning/ *
      ap_perception/ *
      ap_safety_monitor/ *
      ap_srvs/
      ap_tasks/ *
      ap_test/
    deploy/
```

The directory `acs_ros_ws` is a ROS workspace; `autopilot_bridge` is a ROS package that comes from its own repository, and `autonomy-payload` is a collection of ROS packages and supporting files that comes from another repository. (`autopilot_bridge` was originally part of `autonomy-payload` but was later separated as part of its open source release.)
Note that there is a one-way dependency of **autonomy-payload** on **autopilot_bridge**; software in **autopilot_bridge** MAY NOT depend on any resources in **autonomy-payload**.

There are additional directories in the user's home directory that contain libraries, configuration files and logs:

```
~/
  acs_ros_ws/  # Workspace installed here on planes only
  bags/
  blessed/
  mavlink/
```

The **bags** directory contains ROS "bag" files, containing recorded ROS topic data. The **blessed** folder contains "blessed" (known-good) autopilot configurations, which the payload can optionally download from a server at startup and use to configure autopilot parameters, waypoints, and so on. The **mavlink** folder contains the MAVLink library, which defines autopilot messaging.

### Core packages

The payload software is divided into several ROS nodes that communicate with each other using ROS topics, services, and parameters. A brief overview of each of the "core" packages and nodes that comprise the payload is given below.

#### autopilot_bridge

The autopilot bridge, also referred to as "mavbridge", is a bridge between the ROS-based payload software and the MAVLink-speaking autopilot. It is similar in functionality to the ROS package **mavros**, though is implemented entirely in Python and implements a number of payload-specific capabilities.

All payload traffic to the autopilot is routed through mavbridge. With the exception of the bridge's MAVLink connection slaving, all traffic is routed from the autopilot to other ROS nodes, or vice versa.

#### ap_safety_monitor

The safety monitor implements basic aircraft health checks. Its original, and primary, function is to periodically send a heartbeat message through mavbridge to the autopilot. If the autopilot does not hear this heartbeat without some timeout, it assumes that the payload has failed, and enters a failsafe mode. By generating the heartbeat in a node other than mavbridge, failures in ROS messaging are also detected.

Whether the safety monitor emits a heartbeat is controlled by a number of safety checks:

- Whether the user has explicitly disabled the heartbeat (for testing)
- Whether mavbridge is emitting pose data (detects one-way communication failures)
- Whether all "critical" ROS nodes are online
ap_network_bridge
In some ways the core component of the payload, the network bridge connects the payload with other aircraft as well as ground stations. It defines event handlers for inbound ROS and network messages as well as time-based events. Except for slaved MAVLink connections that are routed directly through mavbridge, all network traffic passes through the network bridge.

ap_perception
This package contains two nodes that track the position and state of other aircraft; one for "blue" (same-team) planes, and another for "red" (opposite-team) planes.

swarm_tracker_node.py
The swarm tracker maintains awareness of other same-team aircraft. It ingests both pose and status information that all aircraft periodically broadcast, and produces a periodic time-synchronized "snapshot" of all aircraft positions based on last-known position, attitude, and velocities.

red_tracker.py
The red tracker works much like the swarm tracker, but for the opposite team (when playing a two-swarm game). By default, it uses "virtual sensing" based on positions that are re-broadcast between the two swarm networks. However, it may in the future ingest poses generated by computer vision or other onboard or ground-based sensing.

ap_mission_planning
The mission planning package is the heart of the "swarming" capability of the payload. It implements a control framework for starting up, running, and terminating swarm behaviors. These behaviors are implemented as nodes in the ap_path_planning package. More details can be found in controller.md.

ap_tasks
This package implements the analog of a Linux init system for the payload. It detects various system events, from software startup to flight ready status being set (see preflight.md) to aircraft throttle arming. Several things may need to occur upon these events, including downloading "blessed" configuration files, starting logging software, and setting wireless radio transmit power (see wireless.md).

Tasks are derived from class Task, and each class can define handlers for each system event. The order in which task handlers are executed is based on the order in which class instances are created and registered in the main portion of task_runner.py. Note that this means the relative order of handler execution is the same between two tasks across all system events.
Libraries

There are also a number of supporting libraries in the directory `ap_lib`; these are used not only by the payload but also by offboard software that interacts with the payload. For example, the current network communications implementation resides here (in the future, this may be relocated to an independent library). Descriptions of a few noteworthy libraries are given below; some others are discussed elsewhere in the documentation.

acs_messages.py

This library contains definitions and serialization code for all network messages. Each message type is implemented as a class that inherits from `Message`, and defines its own byte packing using the Python `struct` library.

bitmapped_bytes.py

This library contains definitions and serialization code for all parsing the `params` field of the `BehaviorParameter` ROS message, `swarm_parameters` field of the `SwarmBehavior` network message, or `data` field of the `SwarmBehaviorData` network message. These message types utilize a byte array to provide for behavior-specific message implementations without the requirement for additional message types (see `controller.md` for details). Each parser type is implemented as a class that inherits from `BitmappedBytes`, and defines its own byte packing using the Python `struct` library.

acs_socket.py

The Socket class and supporting classes implement unreliable air-to-air and air-to-ground communications, and (optionally pseudo-reliable) ground-to-air communications. The reliability mechanism has known limitations and may also contain bugs; it is recommended to only use the unreliable form of communications currently implemented in this library.

acs_network.py

This library implements the core of the network bridge in class `NetworkBridge`. All network, ROS, and timed event handling logic is contained here; the node in `ap_network_bridge` implements specific event handlers for the aircraft payload. The A.M.P.P.S. launcher software (see `launch.md`) also implements a network bridge using this library.

acs_logger.py

This library implements network message to file logging for all network messages. It also provides support for replaying logged messages onto a network. See also `acs_net_logger.py` in the `acs-env` repository.
nodeable.py

The Nodeable abstract class encapsulates much of the functionality of a payload ROS node, and is in fact the basis of several nodes including the swarm controller, both swarm trackers, and all swarm behaviors.

behavior.py

The Behavior abstract class inherits from the Nodeable class. It should be a parent class (either directly or through further inheritance) of all implemented swarm behaviors. See the document controller.md for details.

waypoint_behavior.py

The WaypointBehavior abstract class inherits from the Behavior class. It should be a parent class (either directly or through further inheritance) of all implemented swarm behaviors that are realized by changing the position of the current autopilot waypoint. See the document controller.md for details.

ap_enumerations.py

This library provides several key enumerations implemented as "constant" numbers and dictionaries. This includes the various "states" that the swarm controller can be in, the set of swarm behaviors, autopilot modes, minimum and maximum operating altitudes, and so on.

distributed_algorithms.py

This library provides a set of potentially useful distributed algorithms for possible use.

gps_utils.py

This library provides a functions for dealing with positions specified in latitude/longitude.

quaternion_math.py

This library provides functions for performing common quaternion operations.

Messages and services

The ap_msgs and ap_srvs directories contain payload-specific ROS message and service definitions. Note that autopilot_bridge defines its own messages and services as well, which some autonomy-payload packages utilize.
Swarm behaviors

The controllers that implement swarm behaviors are contained in `ap_path_planning`. See the document `controller.md` for details.

Launch files

The payload software can be started either manually or automatically (see `deploy.md` for how automated startup is done). In both cases, the ROS software is started using `roslaunch`. There are two ROS launch files in `ap_master/launch/` of `autonomy-payload:master.launch` that launches the software on an actual aircraft, and `sitol.launch` that wraps `master.launch` and supports a desktop simulation of the aircraft with actual autopilot and payload code running.

On the aircraft, there are a few variables that can be passed as command line arguments to the payload software when `roslaunch` is run:

- **id** - the ID of the aircraft (valid range is currently 1-223)
- **name** - the friendly name of the aircraft (usually the hostname of the payload)
- **gps** - Boolean that indicates whether GPS signal is available; if 1, `autopilot_bridge` waits for a valid `SYSTEM_TIME` MAVLink message to set the payload clock
- **do_bag** - Boolean that indicates whether to turn on `rosbag` recording (see below)
- **do_verify** - Boolean that indicates whether to fetch and verify autopilot configuration from a central configuration server (see `mission.md`)
- **ap_dev** - The device to use for communicating with the autopilot
- **net_dev** - The interface to use for communicating with other aircraft and ground stations
- **net_port** - The UDP port to use for network communications

The SITL launch file provides a slightly different set of arguments:

- **sitol** - Maps to `ap_dev`, usually takes a network specification to the running autopilot SITL instance
- **dev** - maps to `net_dev`
- **port** - maps to `net_port`
- **ns** - ROS namespace to use for SITL (default `/`)
- **bag** - maps to `do_bag` (default 0)
- **verify** - maps to `do_verify` (default 0)

Note that the SITL startup scripts (see `sitol.md`) use these arguments in specific ways to achieve the multi-SITL environment on a single machine.

Unit testing

Some preliminary work has been done using the `rotest/unitest` framework. See `ap_test` for an example (note: the current code may not be compatible with the most recent version of the safety monitor node).
Logging and aggregation

ROS bags capture messages sent via topics (publish-subscribe relationships). There are also ROS logs that capture text log messages created using the `rospy.log*` methods. These are stored in `~/.ros/log/` under folders corresponding to each run on the payload.

Due to I/O performance limitations, topics that publish a lot of messages or large messages may need to be excluded from ROS bag recording. This is done inside `task_runner.py`. An example is the `/network/recv_pose` topic, which for a full-size swarm, may publish on the order of 500 messages per second.

In order to capture data that is otherwise not logged or that is excluded, nodes can be created in `ap_logging`. Currently, there is a node, `recv_pose_rate.py`, that records aggregate statistics about the pose messages received from other aircraft (to summarize information that would be derivable if `/network/recv_pose` were directly recorded).