ACS - Controller

This document briefly describes the ACS behavior controller architecture and provides instructions for defining new behaviors.

Behavior object inheritance hierarchy

User-defined behaviors must inherit from either the abstract Behavior or WaypointBehavior class in order to function properly within the ACS behavior architecture. The inheritance structure for these classes is as follows:
A behavior definition template is provided in the `ap_path_planning` package (`behavior_template.py`). User-defined behaviors can be implemented by providing definitions for the stub methods within this template as described in this document.

An instantiation of the `SwarmController` (the `swarm_control` node) class depicted in the UML diagram controls the safe activation and deactivation of user-defined behaviors.

Descriptions of the the abstract classes is provided below:
Nodeable

Provides functionality for an object that can be implemented as a ROS node or as an object within a ROS node.

**Partial list of class variables** (commonly required by subclasses)

- *nodename* - the name of the object's ROS node
- *timer* - the ROS rate that controls the node's timing loop (if the object is in a dedicated node)
- *DEBUG_PRINT* - set to true to print ROS debug messages to the screen (default False)
- *INFO_PRINT* - set to true to print ROS information messages to the screen (default False)
- *WARN_PRINT* - set to true to print ROS warning messages to the screen (default False)

**Partial list of class methods** (commonly required by subclasses)

- *createPublisher* - creates a ROS publisher object to the specified message topic (i.e., node_name/topic_name).
- *createService* - creates a ROS service implementation (i.e., for node_name/service_name).
- *createServiceProxy* - creates a ROS service proxy for the specified service (i.e., for node_name/service_name).
- *createSubscriber* - creates a ROS subscription to the specified message topic (i.e., node_name/topic_name).
- *log_debug* - generate a ROS debug message
- *log_info* - generate a ROS information message
- *log_warn* - generate a ROS warning message
- *runAsNode* - sets up all required ROS functionality and runs the object in a timed-loop ROS node

**Virtual methods that require subclass implementation** (if required by the subclass)

- *callbackSetup* - set up all required class-specific ROS subscriptions and callback methods
- *executeTimedLoop* - Perform all actions required during a single iteration of the objects timed loop
- *publisherSetup* - set up all required class-specific ROS message publishers
- *serviceSetup* - set up all required class-specific ROS services that will be provided
- *serviceProxySetup* - set up all required class-specific ROS service proxies for object use

**Behavior**

Provides functionality for a generic behavior relying on an arbitrary control mechanism. This class should be used to implement behaviors that rely on control mechanisms other than movement of a loiter waypoint (as implemented by WaypointBehavior).

**Partial list of class variables** (commonly required by subclasses)

- *ap_wpt* - ID of the current waypoint in use by the autopilot
- *behaviorID* - the unique integer ID for the running behavior object
- *crashed_keys* - IDs of swarm UAVs suspected of crashing (if subscribed to the swarm_tracker/swarm_uav_states topic)
- *reds* - records for all red UAVs (if subscribed to the red_tracker/red_uav_states topic)
- *reds_lock* - reentrant lock for use in ensuring thread safety when accessing the *reds* variable
- *swarm_keys* - IDs of all subswarm (if subscribed to the swarm_tracker/swarm_uav_states topic)
- *swarm_lock* - reentrant lock for use in ensuring thread safety when accessing the *swarm* variable
- _subswarm_id - the subswarm to which the UAV is currently assigned
- _swarm - records for all UAVs in the swarm (if subscribed to the swarm_tracker/swarm_uav_states topic)
- _behaviorDataPublisher - ROS publisher for sending SwarmBehaviorData messages to other UAVs
- _swarm_keys - IDs of all swarm (if subscribed to the swarm_tracker/swarm_uav_states topic)

Partial list of class methods (commonly required by subclasses)

- set_active - used to safely activate or deactivate the behavior (automatically implemented by the run service)
- set_pause - used to safely pause or resume the behavior (automatically implemented by the pause service)
- set_active - used to set the behavior parameters (automatically implemented by the set service)
- subscribe_to_reds - subscribes to the red_tracker/red_uav_states topic (call in the set_behavior method if rqd)
- subscribe_to_swarm - subscribes to the swarm_tracker/swarm_uav_states topic (call in set_behavior method if rqd)

Virtual methods that require subclass implementation (if required by the subclass)

- _process_swarm_data_msg - callback method for processing behavior-specific inter-vehicle communication. SwarmBehaviorData network messages received from other UAVs. Upon receipt, the network node publishes the contents of all received SwarmBehaviorData messages as ROS BehaviorParameters messages to the network/recv_swarm_data topic (all behaviors automatically subscribe to this topic and register this method as the callback).
- run_behavior - runs a single iteration of the behavior's control loop
- set_behavior - implements the ROS "set" service to initialize the behavior to run
- _safety_checks - conducts any required behavior-specific safety checks

NOTE: the Behavior class implements the Nodeable.executeTimedLoop method. Subclasses should not override this implementation.

WaypointBehavior

Provides functionality for a generic behavior relying on the movement of a loiter waypoint.

Partial list of class variables (commonly required by subclasses)

- wp_msg - message object for holding waypoint location info (lat, lon, rel_alt) to be sent to the autopilot
- _last_wp_id - ID of the last waypoint in the autopilot waypoint list (must be an infinite loiter waypoint)

Partial list of class methods (commonly required by subclasses)

- publishWaypoint - publishes the values from the wp_msg variable to the autopilot

Behavior activation
Behavior registration and activation overview

User-defined behaviors must be self-contained (i.e., they cannot operate through other running behaviors) and must run in a dedicated ROS node. Upon startup, each swarm behavior node uses the `swarm_control/register_behavior` service to register with the `swarm_control` node under its unique `behavior_id`. This transaction is implemented in the `Behavior` class and occurs automatically upon behavior node startup. The UAV is ready for swarm behavior initiation upon initial arrival at the standby waypoint.

Behavior activation and deactivation is controlled by the `swarm_control` node. Among other things, this node ensures the following behavior activation semantics are observed:

- The "swarm ready" state is set following launch, climbout and transition to the standby waypoint.
- UAVs can only be assigned to subswarms other than '0' in the swarm ready state.
- A UAV's subswarm assignments can only be changed while the "standby" behavior is active (i.e., no user-defined behavior is in progress), and the subswarm to which the UAV is being assigned cannot have a user-defined behavior active.
- User-defined behaviors can only be activated when the UAV is assigned to a non-0 subswarm.
- User-defined behaviors can only be activated when the UAV is in the "swarm ready" state.
- UAVs can only be assigned to user-defined behaviors that have not failed (i.e., that are regularly reporting their status).
- UAVs can only be assigned to user-defined behaviors when the autopilot is in AUTO.
- User-defined behaviors inheriting from the `WaypointBehavior` class can only be activated if the last waypoint in the mission file is an infinite loiter waypoint.

Behavior activation sequence

The ACS behavior activation sequence depicted in the following diagram:
Behaviors activation is initiated by receipt of a SwarmBehavior message over the network. Upon receipt, the
network node converts the message content to an `acs_msgs/BehaviorParameters` message uses the `swarm_control/run_behavior` service to initiate activation.

If the requested behavior is the "standby" behavior, all user-defined behaviors are deactivated by the `swarm_control` node using their respective `run` services and the UAV is ordered to the standby waypoint. For other behavior types requested behavior ID is tested against the `swarm_control` nodes behavior registry. If the requested behavior is valid, the following safety checks are conducted:

- Is the UAV in the "swarm ready" state?
- Is the UAV assigned to a taskable subswarm (i.e., not subswarm 0)?
- Is the requested behavior running (still reporting its status)?
- Is the autopilot in AUTO mode?
- If the behavior uses waypoint control, is the last waypoint in the list an infinite loiter waypoint?

If the safety checks are successful, the `BehaviorParameters` data is forwarded to the requested behavior's `set` service. If they are unsuccessful then the `swarm_control` node behaves as if it received a "standby" behavior request.

Upon invocation, a behavior's `set` service will use a behavior-specific `BitmappedBytes` object to parse the requested parameter values from the ROS service request's `params` field. The `set` service will set the behavior parameter values according to the request and set the behavior's ready state to "True" if they are valid ("False" otherwise). The behavior's ready state is then returned as the service response.

If the behavior's `set` service response indicates that the behavior was successfully set (i.e., its ready state is "True"), the `swarm_control` node will deactivate all other behaviors using their respective `run` services and then utilize the requested behavior's `run` service to activate the behavior. If the behavior was not set to the "ready" state, then the `swarm_control` node will behave as if it received a "standby" behavior request.

Behavior activation relies on the ROS communications architecture depicted in the following diagram:
All depicted ROS services, message topics, publishers, and subscribers are set up automatically during system initialization.

**User-defined behavior implementation steps**

1. Choose a unique integer to use as the behavior ID for the new behavior. This value can be optionally added as a constant to `ap_lib/acsEnumerations.py`. The behavior node should be initialized with this value at instantiation, and the `swarm_behavior` field of `SwarmBehavior` messages should be set to this value to invoke the behavior.

2. Determine the appropriate activation parameters (i.e., any values needed to initialize the behavior) and implement a `BitmappedBytes` subclass to parse the required parameters from an array of bytes (or use an appropriate existing subclass). This parser can be used to pack the `swarm_parameters` field of a `SwarmBehavior` network message for behavior invocation or to unpack the `params` field of the ROS `BehaviorParameters` received by the behavior's `set` method.

3. Determine any required behavior-specific inter-vehicle messaging requirements and implement `BitmappedBytes` subclasses to parse the values from an array of bytes (or use an appropriate existing
subclass). Unique integer IDs must be utilized for each type of data message (the id field of the
associated ROS BehaviorParameters message will be set to this value). These parsers can be used to
pack the params field of messages to be sent to other UAVs or parse the params field of messages
received from other vehicles.

4. Use `ap_path_planning/behavior_template.py` as a template for behavior object
   implementation.

5. Stub methods are provided in `behavior_template.py` for all WaypointBehavior class virtual
   methods. Virtual methods only need to be implemented if their functionality is required by the user-
   defined behavior. The following guidance is provided for virtual method development:

   - `callbackSetup` (may be required) - This method is used to set up behavior-specific ROS
     subscriptions and associated callback methods. The Nodeable class createSubscriber method can
     be used to create each subscription. The ROS topic from which the subscriber will receive
     messages will be named according to the following convention: `node_name/topic_name`. This
     can be remapped in the launch file (`ap_master/master.launch`) as required. This method is
called automatically by the `run_as_node` method.

   - `_process_swarm_data_msg` (may be required) - This method processes inter-vehicle
     communications that affect behavior execution (implementation is not required if the behavior
does not rely on inter-vehicle communications). This method should use an appropriate
     BitmappedBytes subclass instance (from step 3 above) to parse the BehaviorParameters message
     params field and modify the active behavior's settings accordingly. If the behavior's active state
     is "False", this method should ignore any received messages.

   - `publisherSetup` (may be required) - This method is used to set up behavior-specific ROS
     publishers. The Nodeable class createPublisher method can be used to create each publisher.
     The ROS topic to which the created publisher will publish messages will be named
     according to the following convention: `node_name/topic_name`. This can be remapped in the
     launch file (`ap_master/master.launch`) as required. This method is called automatically by
     the `run_as_node` method.

   - `run_behavior` (always required) - This method runs one iteration of the behavior's control loop.

   - `serviceSetup` (may be required) - This method is used to set up behavior-specific ROS services
     and associated handler methods that will be provided by the behavior node. The Nodeable class
     createService method can be used to create each service. The ROS service will be named
     according to the following convention: `node_name/service_name`. This can be remapped in the
     launch file (`ap_master/master.launch`) as required. This method is called automatically by
     the `run_as_node` method.

   - `serviceProxySetup` (may be required) - This method is used to set up behavior-specific ROS
     service proxies that are required by the node. The Nodeable class createServiceProxy method
     can be used to create each service proxy. The ROS service that the proxy invokes will be
     named according to the following convention: `node_name/service_name`. This can be remapped
     in the launch file (`ap_master/master.launch`) as required. This method is called automatically
     by the `run_as_node` method.

   - `set_behavior` (always required) - This method implements the behavior's set service. If the
     behavior requires access to swarm UAV states and/or red UAV states, the Behavior class
     subscribe_to_swarm and/or subscribe_to_reds methods respectively should be called by this
     method's implementation. The BehaviorParameters message's params field should be parsed by
     the appropriate BitmappedBytes subclass (from step 1 above) to parse and set behavior
     initialization values. If the behavior parameters are set to valid values, the behavior's set_ready
     method should be used to set the ready state to "True" (use the method to set the ready state to
     "False" if the parameters are not valid). The method should return the post-initialization ready
     state.

   - `safety_checks` (usually required) - This method conducts any behavior-specific safety checks
     and returns "True" if they are successful. The method is called automatically immediately
before the run_behavior method is called. This method is only required to determine the status of the safety checks—the Behavior class implementation will automatically deactivate the behavior if the checks are unsuccessful (return "False").

- set_pause (may be required) - This method controls the vehicle behavior when the user-defined behavior is active but paused. By default, when a user-defined behavior is paused it simply stops issuing commands (meaning that the last pre-pause command will remain in force). User-defined behaviors inheriting from the WaypointBehavior class will thus proceed to the last ordered waypoint position and orbit. If another pause behavior is desired (e.g., shifting the loiter waypoint position to the current vehicle position) it can be implemented here.

6. Instantiate the object and run it as a node as depicted in the main function of
behavior_template.py, changing the node_name, behavior_id and hertz values as required.

7. Add an entry to ap_master/master.launch to launch the new behavior as a ROS node upon system startup. Include "remap" entries to route any topics or services instantiated with the Nodeable class create methods to the correct topic namespace as required (default is nodename/topicname, which may need to be remapped to a different basename).