ACS - Arbiter Multi-Swarm Game Engine

This document describes the concept and design of the Arbiter - ground software that acts as an intermediary between two swarms that are flying in a competitive ("game") context.

Game concepts

The current concept of a two-swarm game is approximated as a cross between aerial laser tag and aerial capture the flag. There are three primary ways of scoring points: successfully launching and flying for a duration, successfully landing within a designated landing zone, and successfully tagging aircraft in the other swarm.

Aircraft pose and status can be used to evaluate launch, flight duration, and landing location. In the absence of actual lasers and laser detectors or another tagging technology, tagging can be done virtually by transmitting "fire" messages over the network, and having a virtual referee evaluate those fires in light of broadcast pose data from both swarms.

Arbiter facilities

To support game play, the Arbiter should support several functionalities that are discussed below. Note that not all of these features exist in the current version of the Arbiter.

Scoring

The Arbiter is responsible for evaluating and maintaining a score for each aircraft and each swarm as a whole. Evaluation is made based on network messages received from aircraft, as well as any external sensor data that is available.

Initially, the Arbiter ingests *Pose* and *FlightStatus* messages to infer takeoff, flight time, and landing time and position. It additionally ingests *SwarmBehaviorData* messages with the *FIRING_REPORT* data type to evaluate tagging. Pose data from the firing aircraft and target aircraft are interpolated to the timestamp of the fire message, and when the firing aircraft is pointed to within an envelope of the target, the target is counted as tagged.

Status reporting

Aircraft will need to know game status information, such as when the game has started, how much time is remaining, which aircraft are "active" (versus penalized or tagged), and whether there are any emergency
conditions requiring aircraft to immediately depart the game airspace.

Additionally, team ground stations and observer stations will need to monitor the above information in addition to team score data.

**Virtual sensing**

In the absence of visual or other on-aircraft sensing, the Arbiter can provide virtual sensing by forwarding **Pose** messages from one swarm as **RedPose** messages to the other swarm. Source IDs and timestamps are preserved so opposing swarms can track and forward-project aircraft positions as they would with shared positions within their own swarm.

**Rule enforcement**

Along with scoring, the Arbiter will enforce any gameplay rules, ranging from informing tagged aircraft so they may leave the game airspace (if appropriate) to commanding aircraft that violate a gameplay rule to go to a virtual "penalty box."

**Logging**

To replay and analyze games, a central log is immensely useful. Of course, due to the constrained and lossy nature of networks, aircraft logs are also invaluable for reconstructing the details of a game.

A module currently exists to record all messages received from both swarms. Console output is logged separately as well.

**Arbiter design**

As with the Swarm Commander and the Health Monitor (see `ground.md`), the Arbiter has a modular design with a central core that maintains state and handles events. It builds on the same core libraries in `acs_lib` as other major ground software.

**Core files**

- **arbiter_start.py** - This is the runnable file that is installed in the path. Its main role is to register modules (see below), create and initialize an **ArbiterState** object, and run an interminate loop to handle "events" that arise from the core and modules.

- **arbiter/arbiter_state.py** - This is the core of the Arbiter. It handles loading modules as objects (see below), processes command line arguments, and establishes handling of time and event-driven callbacks. It also stands up network communications and initializes the internal state for both swarms (the "snapshots").
Swarm snapshots

The way that aircraft state are maintained by the Arbiter bears some extra attention.

*FlightStatus* messages are stored in a straightforward manner: for each aircraft that has been heard from since the Arbiter started, an *ACS_UAVState* object is created. The last-heard *FlightStatus* message is always held by that object.

*Pose* messages are stored in a *MultiScroll* object, which is effectively a collection of queues with certain additional properties. The MultiScroll limits the amount of pose data that it holds, in order to limit memory use. However, it also attempts to retain sufficient pose data to interpolate poses for fire message evaluation. It further provides facilities to ensure that data past a certain "safe" threshold (in time) are not removed; this is used to ensure that there exists pose data for the time period between the threshold of when all fire message have been evaluated and the present time. More details can be found in *multiscroll.py*.

Modules

As noted above, all Arbiter modules derive from the *ArbiterModule* class in *arbiter_module.py*. Modules may define callbacks that fire when blue and red team messages arrive, on a periodic basis, and when command line arguments are being processed. They also have access to the *ArbiterState* singleton through *self.state*.

At the time of this writing, three modules have been written:

*arbiter/arbiter_logger.py* - This module logs all messages received from aircraft as well as ground stations, per swarm. These are logged to file in a format defined by *LogWriter* in the *autonomy-payload/ap_lib* library *acs_logger.py*.

*arbiter/arbiter_score/* - This module defines the scoring engine, which evaluates fire messages as well as takeoffs and landings.

*arbiter/arbiter_sensor.py* - This module provides the virtual sensor, forwarding pose data between swarms.
API - Inputs and Outputs

The current aircraft payload software is designed to interface with the current Arbiter, with the limitations discussed below. For independently designed swarms to interface with the Arbiter, the following API must be supported by the swarm or a proxy on its behalf:

Inputs to Arbiter:

- **Pose** - All aircraft must emit pose data at a "reasonable" rate (nominally, several times per second) to support scoring and rule enforcement.
- **FlightStatus** - The flight status message is currently a catch-all that is specific to the onboard payload design. However, its **armed** (flag indicating throttle is armed), **airspeed** (in m/s), and **alt_rel** (altitude relative to takeoff point in mm) fields are used for takeoff and landing time determination.
- **SwarmBehaviorData** - This message with the **data_type** field set to **FIRING_REPORT** is used to indicate a fire from one aircraft at another.

Outputs from Arbiter:

- **RedPose** - This is a variant of **Pose** used for virtual sensing between swarms.
- **GameStatus (TBD)** - A message may exist to indicate game status to each swarm, including time elapsed and current score.
- **Hit (TBD)** - A message may exist to indicate to an aircraft or swarm when an aircraft has been tagged by the other swarm.
- **Penalty (TBD)** - A message may exist to indicate to an aircraft or swarm when a gameplay rule has been violated.

Operation

The Arbiter is started from the command line as

```
arbiter_start.py
```

By default, it runs on two virtual network interfaces set up by multi-SITL instances, **sitr_bidge_1** and **sitr_bidge_2** (see **sitr.md** for details). To override these settings for the red and blue teams (note that "red" and "blue" are synonymous with "1" and "2"; there is no semantic distinction by color), use the **-dr** and **-db** arguments. Other options are available; use the **-h** argument for a list.

Limitations and future work

Currently, which takeoff, landing, and tagging are evaluated and recorded, the actual score is not calculated nor is it reported.

Virtual sensing via **RedPose** messages has been implemented and used in live-fly tests. However, no evaluation of network impact has been performed. Further, the latency and ability to forward-project latent red pose data has not been evaluated.
The Arbiter relies on synchronized clocks for fire message evaluation. It is assumed that the computer running the Arbiter and all aircraft have synchronized clocks. However, there is no facility implemented to check and correct for this. See `future.md` for more discussion on clocks.

No **Hit** message exists, and there is no indication to a firing or target aircraft that tagging was successful.

No gameplay penalties have been defined or implemented.

No **GameStatus** message exists, for aircraft, ground systems, or observers.