ACS - Simulation-In-The-Loop (SITL)

This document provides an overview of how the ACS SITL environment is constructed, and how to start up and run one (or multiple) simulated swarms.

**Quick Usage**

To start a single simulated aircraft autopilot and payload, simply run:

```bash
multi-sitl-start.bash -B 1
```

which is equivalent to:

```bash
multi-sitl-start.bash -B -T 1 -I 1 1
```

Option `-B` tells the SITL to use a software bridge for networking (using this will require `sudo` privileges and will cause a virtual bridge named `sitl_bridge_1` to be created). This option is necessary even for single aircraft, in order for ground software to communicate with the simulated payload. Option `-T` specifies the "team ID" for the aircraft, and option `-I` specifies the starting aircraft ID to use; note that 100 is added to this number, so the first aircraft ID is actually 101. The final `1` specifies the number of simulated aircraft to start.

Ground software, such as swarm commander, health monitor, and the flight tech interface, can talk to the simulated payload over `sitl_bridge_1`. For instance, the FTI can be started with:

```bash
fti.py -d sitl_bridge_1
```

When running the FTI with many SITL instances, it is recommended to use the `-z` option to enable "debug" features that make it easier to work with many virtual aircraft. The `-z` option should NOT be used when operating real aircraft!

To start a simulated swarm of 5 aircraft:

```bash
multi-sitl-start.bash -B 5
```

They will come up with aircraft IDs 101-105. To start another swarm (as a different team):

```bash
multi-sitl-start.bash -P -B -T 2 -I 11 5
```

These will come up with IDs 111-115 (`-I 11` implies starting ID of 111); it is important that all aircraft have unique IDs, even across teams (future implementations may change this constraint). The `-T 2` option specifies that this is team number 2; the implications of this are discussed below. Option `-P` is critical: `multi-sitl-start.bash` by default cleans up any stray simulated aircraft before starting the new simulation; `-P` overrides this behavior, allowing additional simulated aircraft to be run on a system that is already simulating aircraft.
The second team's network interface will be `sitl_bridge_2`; ground software can communicate with those aircraft across that interface, and the two swarms will not be able to directly communicate with each other. (An Arbiter instance will provide necessary inter-team communication.)

To stop and clean up all running simulations:

```
multi-sitl-cleanup.bash
```

### Initializing SITL instances

SITL instances must be initialized before being run. Each instance requires a directory in the `~/flying/` directory, numbered according to the aircraft ID. For instance, aircraft with ID 105 requires a directory `~/flying/sitl15/`. In each directory, a copy of `eprom.bin` (the autopilot firmware compiled for the local computer architecture) is generated and configured with default parameters and mission configuration.

The ACS installer (see `install.md`) initializes 3 SITLs, with IDs 101-103. Additional instances can be initialized using `init_n_sitls.bash`:

```
init_n_sitls.bash -I FIRST_ID NUMBER_SITLS
```

where `FIRST_ID` is the lowest numbered ID instance to initialize, and `NUMBER_SITLS` is the number of instances to initialize. To initialize SITLs with IDs 4-8 (5 instances total), run:

```
init_n_sitls.bash -I 4 5
```

### Under The Hood

Within `multi-sitl-start.bash`, a few other scripts are called or sourced to start up the autopilot, payload, and network pieces of the SITL instance(s).

### sim_vehicle.sh - Simulated Autopilot

This piece is developed and provided by the Ardupilot community; it compiles and runs the autopilot code using a Hardware Abstraction Layer that allows it to run on a normal computer with a flight dynamics model (FDM) in place of actual autopilot hardware and avionics.

It runs an instance of the JSBSim FDM to provide flight simulation. The FDM accepts control surface commands and provides GPS and IMU inputs to the running autopilot code.

This script also starts an instance of MAVProxy, a command-line ground control station, to provide a virtual telemetry link to the simulated aircraft over TCP/IP, and to allow the user to command the aircraft from within MAVProxy. This piece is necessary for simulated autopilots to communicate with simulated payloads.
launch_payload.sh - Simulated Payload

This script wraps the startup of the autonomy payload software (which is ROS-based), with a variety of options, including network device and port selection, as well as support for Linux containers. For the typical SITL use case, however, most of these options are unneeded; the `-D` device selection option is used to run the payload using the `sitr_bridge_*` network device. A numeric argument specifies the ID for the aircraft (e.g., 101).

launch_lib.sh - Virtual Network Bridge

While `launch_payload.sh` is capable of standing up a virtual network bridge on its own, to facilitate multiple simulated aircraft sharing a common bridge device, the bridge setup functionality is contained in its own "library" script, which is sourced by `multi-sitl-start.bash`.

Bridging is set up using the Linux `ip` command. The bridge is assigned a name of `sitr_bridge_TEAM` (e.g., `sitr_bridge_1` for team 1), and an IP address of `192.168.(TEAM+1).250` (e.g., `192.168.2.250` for team 1). All simulated aircraft use this device directly; thus their IP addresses are also `192.168.2.250`, for instance.

Networking Considerations

If running two teams on a single machine, the first team will use `sitr_bridge_1` with an IP of `192.168.2.250`, and the second team will use `sitr_bridge_2` with an IP of `192.168.3.250`.

Each team requires its own instance of each ground software program. For example, the flight tech interface run on `sitr_bridge_1` (team 1) will not be able to communicate with aircraft in team 2.

An Arbiter instance can be run on the same local machine, and configured to use the two virtual bridge interfaces to communicate with the two teams.

Alternatively, the multi-team simulation can be spread across multiple machines. This can be done quite arbitrarily, in fact, with one or more simulated aircraft within a team per machine, as long as all members of a single team are able to communicate via broadcast traffic with one another (e.g., via a wired or wireless interface). The second team can be configured similarly but on a separate broadcast domain, and the two team can be linked by an Arbiter instance running on a machine that is connected to both broadcast domains.

As an example, suppose one wants to create a 30 aircraft swarm, spread across 3 machines. Each machine has a network interface eth0 with an IP in the subnet 192.168.1.0/24, and all machines' eth0 interfaces are connected together. The swarm could be started with the following commands across the 3 machines:

Machine 1, IP 192.168.1.51: `multi-sitl-start.bash -D eth0 -I 1 10`
Machine 2, IP 192.168.1.52: `multi-sitl-start.bash -D eth0 -I 11 10`
Machine 3, IP 192.168.1.53: multi-sitl-start.bash -D eth0 -I 21 10
Note that the -P option is no longer necessary since no subsequent SITLs are started on the same machine.
Also note that aircraft IDs must still be deconflicted, even though the aircraft are spread over multiple machines.

For team 2, the same configuration can be applied on a separate set of machines, with two important caveats. First, the IDs must again be unique among each other and must be unique with respect to those in team 1. Second, the machines should use a different IP subnet (e.g., 192.168.2.0/24) and should not be directly connected to the first set of machines. This keeps team traffic properly separated.

The Arbiter instance should then run on its own machine with two interfaces, such as eth0 with IP 192.168.1.1 and eth1 with IP 192.168.2.1, each plugged into the corresponding network. Additional machines to support more SITLs as well as ground stations can be added as desired. Two wireless networks can also be substituted for wired networks.