ACS - Wireless Networking

This document serves to provide background on the design and current implementation of primary aircraft-aircraft and aircraft-ground wireless communications for Aerial Combat Swarms (ACS). It further serves to document several of the technical challenges and solutions encountered in the course of developing the current implementation.

The primary communications system for ACS is a 2.4 GHz 802.11n ad hoc wireless network, connected on each aircraft to the secondary controller (as referred to as the "companion computer" or "payload"). All communication is performed using discrete messages, which are serialized and sent using a custom protocol built on top of UDP and IPv4.

Aircraft also carry a 900 MHz "telemetry" radio link and a 2.4 GHz "RC" radio link, both connected directly to the aircraft autopilot. These links are out of scope of this document.

Design Considerations

The purpose of the communications system is to facilitate messaging between aircraft and ground stations. Types of messaging and their associated requirements include:

- Commands from ground stations to individual aircraft or groups of aircraft
  - Delivery must succeed, some ordering of messages may be required
  - Generally event-driven vice periodic
- Telemetry (e.g., pose and status) from aircraft to other aircraft and ground stations
  - Generally periodic (e.g., 10 Hz pose messages, 2 Hz status messages)
  - Loss of messages is permissible, within statistical percentages and maximum in-a-row
- Swarm behavior coordination from aircraft to other aircraft
  - Loss may or may not be permissible, depending on coordination algorithm
  - Messages may be periodic or event driven, depending on coordination algorithm

From these messaging requirements and the structure of the ACS construct, some general requirements for a communications system are evident:

- Scalable to approximately 60 nodes (50 aircraft plus several ground stations)
- Effective in a lossy RF environment with changing connectivity due to fast-moving nodes
- Allows communication from any node to any or all other nodes
- Supports but does not mandate delivery guarantees
- Low latency in delivering messages

The remainder of this section lays out some of the design considerations for satisfying these requirements.

Network Structure
Since all network nodes must be able to communicate with all other network nodes, some form of multiple-access network is necessary. An alternative would be a collection of point-to-point links to one central message router; however, this is generally too expensive and cumbersome in practice.

Most available wireless networking solutions utilize a single channel; that is, if all network nodes are in communications range of each other, only one node can effectively transmit at once. Multi-channel solutions exist but tend to be more expensive and more difficult to physically integrate with small aircraft.

Both centralized (e.g., infrastructure) and decentralized (e.g., ad hoc) systems may be considered. Centralized systems may offer overall capacity advantages since the master (or access point) can coordinate communications, reducing the overhead of a multiple-access protocol. However, centralized systems are generally designed so that nodes transmit each message to the master, which then retransmits the message to the receiving node(s). This approach utilizes additional capacity (two times the airtime) and introduces added latency. It further requires either that all nodes be within range of a single master, or deploying a multi-master architecture to ensure that each node is within range of at least one master and connecting all masters via some backhaul communications system. Since a considerable fraction of messaging is aircraft-to-aircraft, a decentralized or ad hoc approach may provide better performance.

Mesh networks, also referred to as mobile ad hoc networks or peer-routing networks, utilize all network nodes to relay messages between other nodes. The advantage is that two nodes which are out of direct communications range from one another can use intermediate nodes as relays for their messages. For decentralized networks covering a large area, this may be of significant benefit. However, mesh networks have certain costs and limits that should be considered:

- All nodes must send additional control messages to discover the network topology
- The mesh may not be able to re-form as fast as the topology changes with fast-moving nodes
- Each retransmission of a message consumes the wireless channel and adds delivery latency

Whether or not a mesh network makes sense depends on how frequently nodes (aircraft) are out of range of one another and the ground stations and on how tolerant the messages are to loss or latent delivery. For critical command messages from the ground, a mesh improves the likelihood of delivery to aircraft that are far away. However, for periodic status updates from aircraft in a topology where aircraft are frequently but momentarily out of range of one another, meshing may incur a higher cost than the benefit it delivers.

**Network and Transport Protocols**

The core capabilities that the network can provide to the messaging system are addressing, error detection, and certain delivery guarantees.

Some messages, particularly command and coordination messages, must be addressed to either a single aircraft or a group of aircraft. Pose and status messages may be addressed to all other aircraft and ground stations. Therefore, unicast, multicast, and broadcast addressing is useful. If the messaging protocol itself provides addressing, then another option is to broadcast all messages and let the messaging system filter
received messages accordingly.

Wireless communications are particularly susceptible to message corruption; therefore an error detection mechanism is necessary to detect and either correct or reject invalid messages.

Because of limited wireless signal range and contention between transmitting nodes, many messages will be lost in transit. The communications system should be resilient to lost messages. Since not all messages require guaranteed delivery, the network should provide the ability to send messages without imposing the cost of reliable delivery mechanisms. It may optionally provide a reliable delivery mechanism that can be used selectively, or leave that to the messaging protocol.

TCP is a widely used protocol for reliable delivery of data. Within the limits of the underlying network, it guarantees that all bytes arrive, in order, and within causing network congestion. While these guarantees are useful for many applications, they cause problems in this domain. There is no way to send a message* unreliably with TCP; no subsequent message can be delivered until all preceding messages have been delivered. TCP also treats loss as an indication of congestion and reduces the maximum rate at which messages can be sent; this is counterproductive for periodic and loss-tolerant messaging. Further, TCP is connection-oriented between pairs of nodes, meaning for N nodes, N^2 TCP connections would have to be created and maintained.

* TCP actually treats traffic as a continuous stream of bytes, not as discrete messages. An application protocol can discretize the byte stream into messages.

**Messaging Protocol**

Two major principles that can guide the messaging protocol are:

- **Statelessness** - Messages should in general be atomic; they should not rely on the receipt of other messages. Senders should also make minimal assumptions about the states of receivers.
- **Idempotence** - Any message received more than once MUST alter receiver state AT MOST once. The exception is if an intervening event, including another message, has again altered the state that was altered by the message.

A stateless* design actually lends itself to loss-tolerant messaging. Considering periodic pose and status messages, each receiver can maintain its picture of the world based on the last messages received. If some messages are lost, the receiver can either use the last-received data as-is or extrapolate from that data.

An idempotent design likewise lends itself to implementing reliable messaging on a lossy network. If receivers are able to receive the same message multiple times without adverse effects, a naïve reliability mechanism is to send a message repeatedly until the desired effect is observed at the sender. The use of explicit acknowledgment (ACK) messages is one way for a receiver to indicate receipt to the sender. Negative acknowledgment (NACK), where the receiver node sends a message when it recognizes that a message was lost, is more complicated to implement but can also be useful.
Another method is for the sender to examine the status messages from the receiver. For example, if a command message is sent to an aircraft commanding it to arm its throttle, that aircraft can set a "throttle arming" bit in its periodically-sent status message. The sender would then examine subsequent status messages from that aircraft for that bit to be set. Again, following idempotent design, the sender only checks for a certain state, rather than a specific change in state.

All messages may benefit from some ordering, particularly if multiple messages are being (re-)sent at once or if the underlying network might cause out-of-order delivery. A sender-supplied timestamp or sequence number may be used to resolve message ordering at the receiver.

*Because receivers may maintain state based on received messages, this design approach is sometimes called "soft state."

### Implementation

#### Networking

The current implementation is built on top of UDP over IPv4 over 802.11n in ad hoc mode; no meshing is currently used. UDP provides message error detection, while 802.11n provides the requisite multiple-access protocol. No delivery guarantees are provided; all reliability is implemented by the messaging protocol or by the sending and receiving applications. Although IPv4 implements unicast, multicast, and broadcast addressing, the message protocol explicitly handles addressing, so all messages can be sent as IP broadcasts.

Since 802.11 framing also provides error detection via a checksum (or CRC or FCS) as well as addressing, the messaging protocol could be built directly on top of 802.11n with a reduction in message size, assuming that messages stay within the size limits of an 802.11n frame or the messaging protocol implements its own fragmentation service. In practice, it is easier to implement a protocol on top of UDP.

#### Network Configuration

The current network implementation uses inexpensive, USB WiFi radios on Ubuntu Linux platforms. Radios are statically configured on each system using the following lines in `/etc/network/interfaces`:

```plaintext
auto wlan0
iface wlan0 inet static
    address 192.168.2.123       # Change 123 to unique number in 2..254
    netmask 255.255.255.0
    gateway 192.168.2.1
    wireless-mode ad-hoc
    wireless-essid zephyr
    wireless-channel 6
    wireless-ap 00:11:22:33:44:55
    wireless-tpower 10
```

This defines a Class C network with 254 usable addresses. A dedicated ground station acts as a gateway (router) to external networks; its IP address is 192.168.2.1. All other aircraft and ground stations have
addresses in the range 192.168.2.2 - 192.168.2.254, for a total of 253 devices.

The wireless configuration is for an ad hoc network named "zephyr". It uses 802.11 channel 6, which has a center frequency of 2.437 GHz. Of particular note are the "ap" and "txpower" attributes. The "ap" attribute sets the BSSID, which is formatted as six colon-separated bytes written in hexadecimal (like a MAC address) and identifies the network in which nodes participate. Typically this is derived from the ESSID, but occasionally two endpoints with the same ESSID will be unable to communicate without the BSSID being explicitly set. The "ap" parameter therefore should be set the same for all endpoints in a common network, and distinct from any other independent network.

The "txpower" attribute sets the transmit power. Using many WiFi radios set to high power in close proximity can cause RF saturation, leading to heavy packet loss. It is recommended, at least while aircraft are on the ground or otherwise in close proximity, that their transmit power be reduced (e.g., to 10 dB). In the current software implementation, aircraft boost their transmit power to 20 dB immediately prior to takeoff. Ground systems used for preflying (using the FTI) may also used reduced transmit power; however, all it is recommended that all other ground systems use a higher transmit power.

There is also a "power" attribute that controls device power management; some devices require this feature to be disabled to function properly.

Most Ubuntu systems use Network Manager to configure network devices. Network Manager periodically checks and resets any device that has been manually configured. However, Network Manager defers to the interfaces file. Supplying a configuration as above and then rebooting the computer is one approach to manually specifying a configuration that persists.

The Linux udev subsystem tracks hardware peripherals and assigns them unique names. When a new USB wireless device is connected for the first time, it is given a unique (to that system) interface name, such as wlan0, wlan1, wlan2, and so on. The name is tracked based on MAC address in the file /etc/udev/rules.d/70-persistent-net.rules. Lines can be added to, modified, or deleted from this file to adjust naming. It is also possible to force ANY (single) wireless device to take on the same name (e.g., wlan0) by using the following line (and deleting all other lines referencing "wlan*". Note that this can cause problems if using multiple wireless devices simultaneously:

```
SUBSYSTEM=="net", ACTION=="add", DRIVERS=="*", ATTR{dev_id}=="0x0", ATTR{type}=="1", KERNEL=="wlan*", NAME="wlan0"
```

The Linux wireless subsystem enforces FCC limits on what frequencies and transmit power levels can be set on a device. This may pose problems if intentionally using a high-wattage device. Linux systems maintain a cryptographically signed database of limits per country. It is possible to generate a replacement database with some effort. A quick fix is to tell the subsystem that the device is operating in a country that has more lenient limits: for instance, Bolivia allows transmitting up to 1 Watt (30 dBm) as opposed to the US limit of 100 mW (20 dBm):

```
sudo iw reg set BO
```
sudo iwconfig wlan0 txpower 30

**wifi_config.sh**

A utility exists in the `acs-env` repository to automate wireless device configuration for ground systems. To use it, specify the wireless interface you wish to configure and the last octet of the IP address you wish to use:

`wifi_config.sh wlan0 123`

If necessary, it will prompt for a `sudo` password. Note that the script will automatically tear down any virtual network bridges created by SITL instances, which may have conflicting IP addresses.

By default, it configures the wireless device for the wireless and IP settings described above. An additional configuration set for a second team can be used instead using the `-2` option. In this case, the IP subnet will be 192.168.3.0/24 and a different wireless SSID and channel will be configured.

The default transmit power level assigned is 10; to change this to 20 for example, specify `-P 20`.

A router to an external network can be stood up using the `-R DEVICE` option, which turns on the IP forwarding kernel option and instantiates a NAT iptables rule for `DEVICE`. For instance, if interface `eth0` faces an Internet connection, one could stand up a router (recall the default gateway IP of .1), running at 20 dBm, for the second team configuration, using:

`wifi_config.sh -R eth0 -P 20 -2 wlan0 1`

**Messaging Protocol**

The current ACS message protocol uses a 16-byte header plus a message-dependent payload. The full ACS message (header plus message payload) is the payload of a UDP datagram, generally sent to a subnet broadcast address (e.g., 192.168.2.255) at UDP port 5554. (Note: due to address binding implemented in `acs_socket.py`, sending to unicast and universal broadcast addresses may not work correctly.) The message header definition is as follows:

```
0   8   16   24
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Message type</td>
</tr>
<tr>
<td>Flags / SSID</td>
</tr>
<tr>
<td>Source ID</td>
</tr>
<tr>
<td>Dest ID</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Sequence number</td>
</tr>
<tr>
<td>Acknowledgment number</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Seconds since UNIX epoch</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>Milliseconds since last second</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td>... type-specific payload ...</td>
</tr>
</tbody>
</table>
```

The fields of the header are defined as follows:

- Message type - used to indicate the type of message payload contents
- Flags - high-three bits, used for early version of reliable messaging
- SSID - low-five bits, subswarm ID of aircraft (assignable by command message)
- Sequence number - used for early version of reliable messaging
- Acknowledgment number - used for early version of reliable messaging
- Epoch seconds - seconds since midnight, 1/1/1970
- Milliseconds - milliseconds into current epoch second

Each message type has a defined payload format, which may be fixed or variable length. If variable length, the payload format should include a field to indicate payload size.

IDs are 8-bit, limiting the protocol to a maximum of 256 entities. However, the ID space is further subdivided into entity IDs, subswarm IDs, and a broadcast ID. Entity IDs range from 0-223, though 0 is generally unused in practice; the broadcast ID is 255. Subswarm IDs range from 0-31 per the 5-bit shared Flags/SSID field. Notice that the range 0-31, plus 224, is 224-254. Messages destined for subswarms are mapped into this range; therefore to address subswarm 15, the destination ID should be set to 224+15 = 239.

The current protocol provides a very restrictive form of reliable messaging using the Flags, Sequence number, and Acknowledgment number. This is meant to be replaced with a more general mechanism in a future version of the protocol. Most periodic messages benefit from ordering, specifically being able to detect and discard messages sent earlier than the latest-received. Timestamps are included in the header to facilitate this where explicit sequence numbers are not used; timestamps additionally aid in extrapolating pose data into the future. The inclusion of timestamps in the message header, versus in specific message types, may also be revisited in future versions of the protocol.

**FlightStatus Message**

The definition of the message header and all message types can be found in `ap_lib/src/ap_lib/acs_messages.py` in autonomy-payload. As an example of a currently-used message type and an explanation of its fields, we consider the FlightStatus message (type 0x0).

All message types are implemented as subclasses of the Message class (see payload.md for additional details). For portability reasons, message fields are generally either bits, integers, or string characters; hence some fields' precisions are truncated to accommodate the full value range. The body of the packed FlightStatus message is as follows:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>armed</td>
<td>ok_*</td>
<td>ready</td>
</tr>
<tr>
<td>batt_vcc</td>
<td>airspeed</td>
<td>alt_rel</td>
<td>mis_cur</td>
</tr>
<tr>
<td>behavior</td>
<td>UNUSED (formerly behavior state bits)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name (aircraft name, 16 bytes)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The FlightStatus message fields are derived from a status message from the autopilot (via autopilot_bridge) combined with swarm controller state and other internal state. Its (packed) fields are
defined as follows:

- **mode** - 4-bit enumeration of autopilot modes *(Status.msg in autopilot_bridge defines the enumeration, which is mirrored in ap_enumerations.py)*
- **armed** - Bit flag indicating whether the aircraft throttle is armed
- **ready** - Bit flag indicating whether aircraft is marked "flight ready"
- **ok_*** - Bit flags indicating the status of various components:
  - ok_ahrs - AHRS health (not actively used at this time)
  - ok_as - airspeed sensor health
  - ok_gps - GPS sensor health and minimum satellites visible
  - ok_ins - Inertial Navigation System health
  - ok_mag - Compass sensor health
  - ok_pwr - Power sensor and battery health
  - ok_prm - Approved autopilot parameters are loaded from payload
  - ok_fen - Approved autopilot geo-fence is loaded from payload
  - ok_ral - Approved autopilot rally points are loaded from payload
  - ok_wp - Approved autopilot waypoints are loaded from payload
- **swarm_state** - 4-bit enumeration of aircraft swarming state *(ap_enumerations.py defines the enumeration)*
- **fence_state** - 2-bit enumeration of the geo-fence state *(Status.msg defines the enumeration)*
- **batt_rem** - Battery percent power remaining * 100
- **batt_vcc** - Battery voltage in millivolts
- **airspeed** - Airspeed in meters/second * 100
- **alt_rel** - Altitude relative to the takeoff point, in decimeters
- **mis_cur** - Current mission (waypoint) number (applies in AUTO mode only)
- **swarm_behavior** - Currently-active swarm behavior (0 indicates no active behavior; the enumeration is defined in ap_enumerations.py)
- **name** - A friendly string name for the aircraft, in ASCII (up to 16 characters)

Performance

Throughput Calculations

Questions about throughput and "bandwidth" come up frequently, and detailed information about 802.11 protocols can be difficult to find. An in-depth discussion is out of scope of this document, but a few details are worth noting here.

Serialization rate and effective throughput are not the same thing. Often, serialization rate is what networking products advertise. It is the speed at which the transmitter sends, *when it is sending*. Effective throughput, on the other hand, is the realizable data rate in a macro sense (i.e., in application data bytes per second).

Throughput figures should account for the overhead of the multiple-access protocol and for all protocol layer headers. 802.11 uses Carrier Sense Multiple Access / Collision Avoidance (CSMA/CA), which mandates certain periods of "dead air" in order to deconflict between transmitting nodes. It also divides the available airtime between transmitting nodes based on demand, so no one node typically gets the full throughput.

Protocol headers and checksums also consume bytes.

Consider the message protocol above. Suppose 50 aircraft sharing their poses at 10 Hz, using the 802.11/IPv4/UDP/ACS protocol stack. Using some back-of-the-envelope math:
802.11 header: 34 bytes (not counting preamble bytes)
IPv4 header:  20 bytes (with no IP options)
UDP header:  8 bytes
ACS header:  16 bytes
Pose payload: 40 bytes (LLA, attitude, and linear and angular velocities)

Full message: 118 bytes

118 bytes/message * 10 messages/aircraft-second * 50 aircraft
= 59000 bytes/second
= 472 kbps (where 1 kbps == 1000 bps)

The network must be capable of sustaining an effective throughput of at least 472 kbps, shared across 50
transmitting nodes, just in order to share pose data. Although an 802.11n ad hoc network may support up to a
54 Mbps serialization rate, that rate is limited by RF conditions, distances between endpoints, et cetera. And
again, the effective throughput will be significantly less due to the multiple-access protocol and header
overhead.

Throughput also depends on packet size. Each message sent entails some overhead due to the multiple-access
protocol, so many small messages will require more overhead more than fewer large messages. The number
of header bytes is fixed, so there is again more overhead with many small messages. However, for a fixed bit-
error rate, smaller messages are less likely to be corrupted in transit than larger messages, mitigating some of
the overhead cost by increasing the probability of successful delivery.

Further Reading

More information on configuring wireless networks in Linux can be found here:

https://help.ubuntu.com/community/WifiDocs/Adhoc


More information on regulatory limits on frequency and transmit power can be found here:


More information on device naming and udev rules can be found here:

http://www.reactivated.net/writing_udev_rules.html