SWARMING WITH CFLIB

Jonas Danielsson @ Bitcraze

BAM days
21 October 2021

Contents

- What is the CFLib?
- What does the CFLib offer for swarming?
- Writing a small swarm controlling program with the CFLib
- Conclusions

CFLib: details

- Attempts to provide mapping to functionality inside of the Crazyflie firmware
- Maintained by Bitcraze, open source, needs your input!
- Uses radio or USB to communicate with the Crazyflie firmware via *CRTP* (**C**razy **R**eal **T**ime **P**rotocol)
- Written in Python
  - Asynchronous by default
- Guide: https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/user-guides/python_api/
- API reference: https://www.bitcraze.io/documentation/repository/crazyflie-lib-python/master/api/cflib/
- Code: https://github.com/bitcraze/crazyflie-lib-python

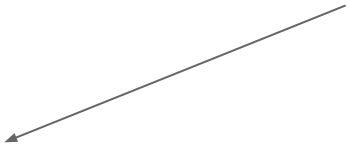
CFLib: the *Crazyflie* module

```
"""
The Crazyflie module is used to easily connect/send/receive data
from a Crazyflie.

Each function in the Crazyflie has a class in the module that can be used
to access that functionality. The same design is then used in the Crazyflie
firmware which makes the mapping 1:1 in most cases.
"""
```

CFLib: asynchronous by default

```python
self._cf = Crazyflie(rw_cache='./cache')

# Connect some callbacks from the Crazyflie API
self._cf.connected.add_callback(self._connected)
self._cf.disconnected.add_callback(self._disconnected)
self._cf.connection_failed.add_callback(self._connection_failed)
self._cf.connection_lost.add_callback(self._connection_lost)
```

```python
def _connected(self, link_uri):
    [...]
    self._cf.log.add_config(self._lg_stab)
    self._lg_stab.data_received_cb.add_callback(self._stab_log_data)
    [...]
```



```python
def _stab_log_data(self, timestamp, data, logconf):
        """Callback from a the log API when data arrives"""
        print(f'[{timestamp}][{logconf.name}]: ', end='')
        for name, value in data.items():
            print(f'{name}: {value:3.3f} ', end='')
        print()
```

CFLib: the *SyncCrazyflie* class

```
"""
The synchronous Crazyflie class is a wrapper around the "normal" Crazyflie
class. It handles the asynchronous nature of the Crazyflie API and turns it
into blocking function. It is useful for simple scripts that performs tasks
as a sequence of events.
"""
```

CFLib: forcing synchronized flow

```
with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
    with SyncLogger(scf, lg_stab) as logger:
        for entry in logger:
            timestamp = entry[0]
            data = entry[1]
            logconf_name = entry[2]

            print(f'[{timestamp}][{logconf_name}]:, end='')
                for name, value in data.items():
                    print(f'{name}: {value:3.3f} ', end='')
                print()
            break
```

CFLib: the *Swarm* class

```
class Swarm:
    """
    Runs a swarm of Crazyflies. It implements a functional-ish style of
    sequential or parallel actions on all individuals of the swarm.

    When the swarm is connected, a link is opened to each Crazyflie through
    SyncCrazyflie instances. The instances are maintained by the class and are
    passed in as the first argument in swarm wide actions.
    """
```

Interlude: Dogfooding

CFLib: a swarm example

```python
with Swarm(uris, factory=factory) as swarm:
    swarm.parallel_safe(activate_high_level_commander)
    swarm.reset_estimators()

    positions = swarm.get_estimated_positions()
    uris.sort(key=lambda uri: positions[uri].y)

    swarm.sequential(take_off)
    time.sleep(5)

    # The `args_dict` arg to the swarm methods expect a dictonary layout:
    #   {
    #       URI: [list, of, params]
    #   }
    #
    # So we turn the positions dictonary from { URI: SwarmPosition } to
    # { URI: [SwarmPosition] }
    parameters = {key: [value] for (key, value) in positions.items()}

    while phase < ITERATIONS * (2 * math.pi / (math.pi / 6)):
        swarm.parallel_safe(wave, args_dict=parameters)
        time.sleep(0.6)
        phase += 1

    swarm.parallel_safe(land, args_dict=parameters)
```

- ~75 lines of Python code
- Flies a swarm in a sinus wave
- Uses the bulk of the CFLib swarm methods

CFLib: a swarm example

```python
with Swarm(uris, factory=factory) as swarm:
    swarm.parallel_safe(activate_high_level_commander)
    swarm.reset_estimators()

    positions = swarm.get_estimated_positions()
    uris.sort(key=lambda uri: positions[uri].y)

    swarm.sequential(take_off)
    time.sleep(5)

    # The `args_dict` arg to the swarm methods expect a dictonary layout:
    #   {
    #       URI: [list, of, params]
    #   }
    #
    # So we turn the positions dictonary from { URI: SwarmPosition } to
    # { URI: [SwarmPosition] }
    parameters = {key: [value] for (key, value) in positions.items()}

    while phase < ITERATIONS * (2 * math.pi / (math.pi / 6)):
        swarm.parallel_safe(wave, args_dict=parameters)
        time.sleep(0.6)
        phase += 1

    swarm.parallel_safe(land, args_dict=parameters)
```

```python
def __init__(self, uris, factory=_Factory()):
        """
        Constructs a Swarm instance and instances used to connect to the
        Crazyflies

        :param uris: A set of uris to use when connecting to the Crazyflies in
        the swarm
        :param factory: A factory class used to create the instances that are
         used to open links to the Crazyflies. Mainly used for unit testing.
        """
        self._cfs = {}
        self._is_open = False
        self._positions = dict()

        for uri in uris:
            self._cfs[uri] = factory.construct(uri)
```

```python
uris = [
    'radio://0/20/2M/E7E7E7E700',
    'radio://0/20/2M/E7E7E7E701',
    'radio://0/20/2M/E7E7E7E702',
    'radio://0/20/2M/E7E7E7E703',
    'radio://1/40/2M/E7E7E7E704',
    'radio://1/40/2M/E7E7E7E705',
    'radio://1/40/2M/E7E7E7E706',
]
```

```python
def __enter__(self):
    self.open_links()
    return self
```

CFLib: a swarm example

```python
with Swarm(uris, factory=factory) as swarm:
    swarm.parallel_safe(activate_high_level_commander)
    swarm.reset_estimators()

    positions = swarm.get_estimated_positions()
    uris.sort(key=lambda uri: positions[uri].y)

    swarm.sequential(take_off)
    time.sleep(5)

    # The `args_dict` arg to the swarm methods expect a dictonary layout:
    #   {
    #       URI: [list, of, params]
    #   }
    #
    # So we turn the positions dictonary from { URI: SwarmPosition } to
    # { URI: [SwarmPosition] }
    parameters = {key: [value] for (key, value) in positions.items()}

    while phase < ITERATIONS * (2 * math.pi / (math.pi / 6)):
        swarm.parallel_safe(wave, args_dict=parameters)
        time.sleep(0.6)
        phase += 1

    swarm.parallel_safe(land, args_dict=parameters)
```

```python
def parallel_safe(self, func, args_dict=None):
    """
    Execute a function for all Crazyflies in the swarm, in parallel.
    One thread per Crazyflie is started to execute the function. The
    threads are joined at the end and if one or more of the threads raised
    an exception this function will also raise an exception.

    For a more detailed description of the arguments, see `sequential()`

    :param func: The function to execute
    :param args_dict: Parameters to pass to the function
    """
```

```python
def activate_high_level_commander(scf):
    scf.cf.param.set_value('commander.enHighLevel', '1')
```

CFLib: a swarm example

```python
with Swarm(uris, factory=factory) as swarm:
    swarm.parallel_safe(activate_high_level_commander)
    swarm.reset_estimators()

    positions = swarm.get_estimated_positions()
    uris.sort(key=lambda uri: positions[uri].y)

    swarm.sequential(take_off)
    time.sleep(5)

    # The `args_dict` arg to the swarm methods expect a dictonary layout:
    #   {
    #       URI: [list, of, params]
    #   }
    #
    # So we turn the positions dictonary from { URI: SwarmPosition } to
    # { URI: [SwarmPosition] }
    parameters = {key: [value] for (key, value) in positions.items()}

    while phase < ITERATIONS * (2 * math.pi / (math.pi / 6)):
        swarm.parallel_safe(wave, args_dict=parameters)
        time.sleep(0.6)
        phase += 1

    swarm.parallel_safe(land, args_dict=parameters)
```

```python
def sequential(self, func, args_dict=None):
    """
    Execute a function for all Crazyflies in the swarm, in sequence.
    The first argument of the function that is passed in will be a
    SyncCrazyflie instance connected to the Crazyflie to operate on.
    A list of optional parameters (per Crazyflie) may follow defined by
    the `args_dict`. The dictionary is keyed on URI and has a list of
    parameters as value.

        Example:
        ```python
        def my_function(scf, optional_param0, optional_param1)
            ...

        args_dict = {
            URI0: [optional_param0_cf0, optional_param1_cf0],
            URI1: [optional_param0_cf1, optional_param1_cf1],
            ...
        }
        swarm.sequential(my_function, args_dict)
        ```

    :param func: The function to execute
    :param args_dict: Parameters to pass to the function
    """
```

```python
def take_off(scf):
    h = get_height(scf)
    scf.cf.high_level_commander.takeoff(h, 5.0)
    time.sleep(5.0)
```

CFLib: a swarm example

```python
with Swarm(uris, factory=factory) as swarm:
    swarm.parallel_safe(activate_high_level_commander)
    swarm.reset_estimators()

    positions = swarm.get_estimated_positions()
    uris.sort(key=lambda uri: positions[uri].y)

    swarm.sequential(take_off)
    time.sleep(5)

    # The `args_dict` arg to the swarm methods expect a dictonary layout:
    #   {
    #       URI: [list, of, params]
    #   }
    #
    # So we turn the positions dictonary from { URI: SwarmPosition } to
    # { URI: [SwarmPosition] }
    parameters = {key: [value] for (key, value) in positions.items()}

    while phase < ITERATIONS * (2 * math.pi / (math.pi / 6)):
        swarm.parallel_safe(wave, args_dict=parameters)
        time.sleep(0.6)
        phase += 1

    swarm.parallel_safe(land, args_dict=parameters)
```

```python
def wave(scf, pos):
    h = get_height(scf)
    scf.cf.high_level_commander.go_to(pos.x, pos.y, h, 0, 1.0)
```

```python
def get_height(scf):
    pos = uris.index(scf.cf.link_uri)
    where = (pos + phase) % 6
    return 1 + 0.6 * math.sin(where * math.pi / 6.0)
```

Demo

CFLib: conclusions

- Meant to get you started with a swarm *quickly*
- Not *meant* for serious research or commercial use
- Python thread model (global interpreter lock)
  - No true concurrency
- No broadcast support

CFLib: what should be in it?

Questions?