

## CRAZYSWARM FRAMEWORK FOR MULTIPLE DRONES

This tutorial is meant to cover the **crazyswarm framework** and how to use it for **synchronised drone flight**. This framework offers all the necessary components for controlling **multiple drones remotely**, by relating the drone flight controller of the Crazyflie to a set of controllers on the PC, but also by offering ways to **send trajectories to the drones in realtime**.

All this **data interlinking** makes use of **ROS** which offers sufficient **system modularity** to use this framework in **unprecedented, creative ways**. This tutorial is therefore directed at **interaction designers** who look to fly multiple drones using their own choice of inputs. The main challenges in swarm robotics are addressed in this framework, namely the **system latency** in terms of system **communication**, and channel testing for the best signal possible.

According to the authors, current limitations to the framework's reliability are the **controller failures**, improving object tracking **failure recovery**, and including better planning methods which take **aerodynamic effects** such as downwash into account explicitly.

### Some key links

The official changelog	<a href="https://crazyswarm.readthedocs.io/en/latest/changelog.html">https://crazyswarm.readthedocs.io/en/latest/changelog.html</a>
Set up (Read the docs yourself!)	<a href="https://crazyswarm.readthedocs.io/en/latest/">https://crazyswarm.readthedocs.io/en/latest/</a>
Theoretical underpinnings:	<a href="http://act.usc.edu/publications/Preiss_ICRA2017.pdf">http://act.usc.edu/publications/Preiss_ICRA2017.pdf</a>
To-the-point powerpoint:	<a href="https://drive.google.com/file/d/15favAyrLLpC_O6nrAp-elbZijFUMLgwV/view">https://drive.google.com/file/d/15favAyrLLpC_O6nrAp-elbZijFUMLgwV/view</a>

## # Why a swarming architecture is required



### Design Motivation

- **Many robots**  
improve robot team size in research validation
- **Indoors**  
controlled setting, no issues with safety and weather
- **Small robot**  
to fit many of them indoors
- **Motion capture**  
vision too CPU-intensive, UWB too noisy
- **Easy to replicate**  
off-the-shelf hardware, open-source code

At its simplest, Crazyswarm attempts to couple an external motion capture technology like Optitrack with the rest of a drone's control loop: knowing its position, the drone will be able to generate and follow a **trajectory more precisely**. When viewing a **single body**, motion capture certainly has **sub-millimeter accuracy**.

However, as the number of drones increases, there are two limiting factors to the drone's control loop: the first is **recognition of the drones** by the optical capture system, and the second is **low communication bandwidth**. Multiple algorithms are therefore incorporated into this framework to mitigate the effects of these processes. These effects will be more thoroughly explored later in this tutorial.

As the number of drones increases, an interesting swarm feature is managing large numbers of drones. Crazyswarm offers command-line tools and a GUI for mass rebooting, firmware updates, firmware version query, and battery voltage checks over the radio. A power-save mode turns off the main microcontroller and LED ring while leaving the radio active, permitting powering on the Crazyflie remotely.

Finally, a ROS framework is utilized on the PC. By broadcasting the drone's pose among other things, ROS is, as stated earlier, an **entrypoint for other software to interact** with the drone's control loop. To reduce latency however, ROS messages are not used in the **critical path between receiving data** from the motion-capture system and **broadcasting estimated poses** to the Crazyflies.

**Step 1: Define your UAV Type**





90 mm, 33 g      120 mm, 124 g      210 mm, 491 g

```
crazyflieTypes.yaml
default:
  bigQuad: False
  batteryVoltageWarning: 3.8 # V
  markerConfiguration: 0
# ...
```

7

**Step 2: List all UAVs**


```
allCrazyflies.yaml
crazyflies:
- id: 1
  channel: 100
  initialPosition: [1.5, 1.5, 0.0]
  type: medium
- id: 40
  # ...
```

- Initial **position** for frame-by-frame **tracking** and **simulation**

8

Interaction designers may want to look at **better understanding ROS** so to use such inputs and outputs conveniently in their code and in their simulations.

➤ [SEE THE ROS AND LINUX BACKGROUND TUTORIAL](#)



**[Linux and ROS Background](#)**  
 ⌚ less than 1 minute read  
 An overview of skills to become a better drone developer.

Note: a Python scripting layer supports development of complex multi-stage formation flight plan. See Trajectory tutorial for this section.

➤ [SEE THE TRAJECTORY TUTORIAL](#)



**[Trajectory Generation \(/3\)](#)**  
 ⌚ less than 1 minute read  
 Developing trajectories for robots to follow.

## ## Recognising the drones

Standard rigid-body motion-capture software such as Vicon Tracker requires a unique marker arrangement for each tracked object. The Crazyflie's small size limits the number of locations to place a marker, making it impossible to form dozens unique arrangements that can be reliably distinguished. In this way, **optical** motion capture reaches a key limitation: when using a **motion capture framework** with **very small robots**, many rigid bodies seem to have identical capture marker arrangements. As a result, the cameras are prone to **confusing the robots from one another**, especially as the robots move past one another – contributing to other forms of occlusion.

Overcoming motion capture's single point of failure will require reimagining a new state estimation system, but it is worth the effort: motion capture can eventually use their **sub-millimeter accuracy** in **multiple UAV scenarios**. In comparison, ultra-wideband radio triangulation shows position errors of over 10 centimeters, too large for dense formations. As for vision-based methods, while they are both accurate and decentralized, the required cameras and computers necessitate a much larger vehicle.

## ## Low communication bandwidth

The second key issue with large drone swarms is communication bandwidth over large numbers of drones. While remote controlled drones are using **frequency hopping** techniques

ur system fully utilizes the vehicles' onboard computation, allowing for robustness against unreliable communication and a rich set of trajectory planning methods requiring little radio bandwidth.

## # The root of the problem: the control loop

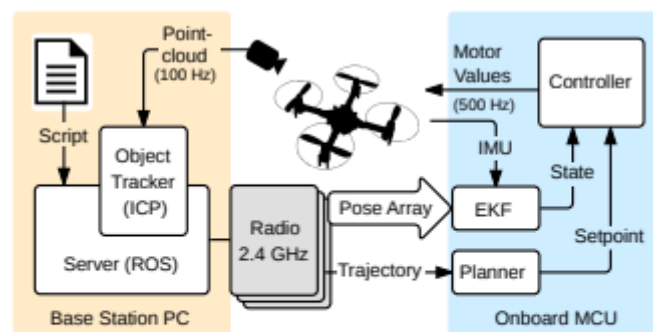


Fig. 2. Diagram of major system components. A point cloud of markers detected by a motion-capture system is used to track the quadcopters. All estimated poses are broadcasted using three radios. Planning, state estimation, and control run onboard each vehicle at 500 Hz.

The main onboard loop runs at 500 Hz. In each loop cycle, the vehicle reads its Inertial Measurement Unit (IMU) and runs the state estimator, trajectory evaluator, and position controller. Messages with external pose estimates arrive asynchronously and are fused into the state estimate on the next cycle.

- As a result of this control loop, a key **criterion** that is used to demonstrate the system's robustness against communication loss during operation is the position error as we reduce the frequency of the **position update rate**.

**TABLE II**  
HOVER POSITION ERROR AS A FUNCTION OF POSITION UPDATE RATE.

Position Rate [Hz]	100	10	5	3.3	2.5	2
Avg. Tracking Error [cm]	0.58	0.68	0.91	1.48	1.95	2.18

- The delay between the immediate onboard IMU response and the corresponding change in external position measurement captures the **full system latency**. We read these values in real time via a JTAG debug adapter.

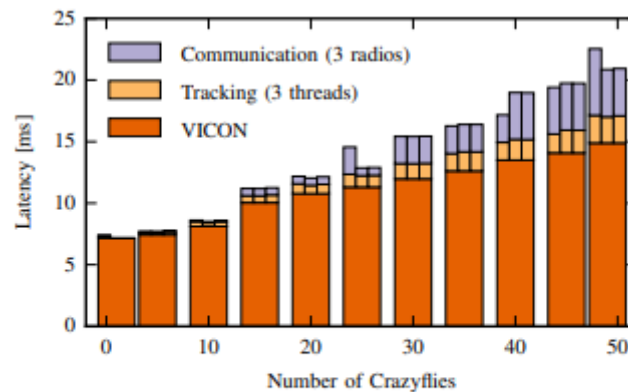
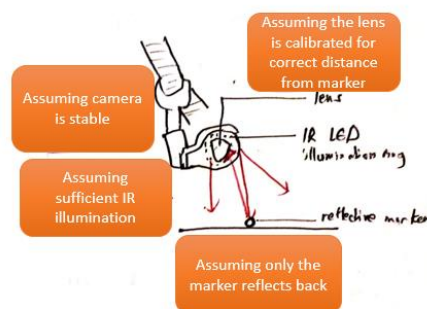


Fig. 3. Software-estimated latency for different swarm sizes, averaged over 2000 measurements. The actual latency is 3 ms higher in all cases. The latency grows linearly with the swarm size and the largest portion is due to the motion-capture system.

ADD THIS

### # Understanding Optical Motion Capture

The most popular forms of motion capture, namely Optitrack or VICON, use a set of technological tools called **optical motion capture**. Optical motion capture uses reflective trackers coupled with infrared sensors. The tracking elements or trackers are small reflective balls. A camera pointed towards the tracker outputs infrared light via its LED illumination ring. The lens at the center of the ring capture any and all IR incoming light.



*There's a lot of assuming.*

There is a large need for a system architecture that can **aid the localizer**.

Crazyswarm mitigates this problem by relying on raw point clouds from the motion-capture system, and implementing its own object tracker based on the Iterative Closest Point (ICP) algorithm that handles identical marker arrangements.

## # Iterative Closest Point

*With identical marker arrangements for each vehicle, the object tracker needs some additional source of information to establish the mapping between vehicle identities (radio addresses) and spatial locations. We currently supply this information with a configuration file containing a fixed initial location for each vehicle. However, it is not feasible to place each vehicle at its exact configured position before every flight. To allow for small deviations, we perform a nearest-neighbor search within a layout-dependent radius about the initial position, and try ICP with many different initial guesses for vehicle yaw. We accept the best guess only if its resulting alignment error is low (less than 1 mm mean squared Euclidean distance between aligned points).*

*Each frame, we acquire a raw point cloud from the motioncapture system. For each object, we use ICP to register the marker positions corresponding to the object's last known pose against the entire scene point cloud. This process is independent for each object, so it can be executed in parallel. This approach assumes that there are no prolonged occlusions during the duration of the flight. We limit ICP to five iterations, which allows operation at 75 Hz with 49 robots using our computer hardware.*

*Motion-capture systems sometimes deliver a point cloud with spurious or missing points; if undetected, this may cause tracking errors. To mitigate these errors, we compute linear and angular velocities from the ICP alignments and reject physically implausible values as incorrect alignments. In combination with our on-board state estimation, a few missing frames do not cause significant instabilities, making tracking reliable in practice.*

## # Communication infrastructure

*Request-response is primarily used for configuration while the Crazyflie is still on the ground. This includes uploading a trajectory, changing flight parameters such as controller gains, and assigning each Crazyflie to a group. Broadcasting is used during the flight to minimize latency for position feedback, and to achieve synchronized behavior for taking off, landing, starting a trajectory, etc. Broadcast commands can be restricted to subsets of the swarm by including a group ID number.*

*Our communication does not use a standard transport layer and hence, we need to handle sporadic packet drops as part of our protocol. In order to achieve low latency, we do not aim for guaranteed packet delivery, but rather for a high probability of reliable communication. In our protocol, all commands are idempotent, so they can be received multiple times without any side effects. This allows us to repeat request-response commands until acknowledged or until a timeout occurs. Swarm-coordination commands, such as taking off, do not wait for an acknowledgement but are repeated several times for a high probability that all Crazyflies receive the command. Since external pose estimates are sent at a high fixed rate every 10 ms, there is no need to explicitly repeat such messages.*

*Empirically, for repeated commands, the likelihood of packet loss depends on the rate at which the command is repeated.*

TABLE I  
CONSECUTIVE PACKETS DROPPED (SEE TEXT FOR DETAILS.)

Delay	0	1	2	3	4	5+
3 ms	54458	2	280	1174	0	0
10 ms	51755	4111	4	1	0	0

*Our communication infrastructure uses compressed one-way data flow and supports a large number of vehicles per radio. We achieve reliable flight with accurate tracking ( $< 2$  cm mean position error) by implementing the majority of computation onboard, including sensor fusion, control, and some trajectory planning. We provide various examples and empirically determine latency and tracking performance for swarms with up to 49 vehicles.*

## Common questions:

### Why use raw point clouds?

Standard rigid-body motion-capture software such as Vicon Tracker requires a unique marker arrangement for each tracked object. The Crazyflie's small size limits the number of locations to place a marker, making it impossible to form 49 unique arrangements that can be reliably distinguished. Therefore, we obtain only raw point clouds from the motion-capture system, and implement our own object tracker based on the Iterative Closest Point (ICP) algorithm [13] that handles identical marker arrangements. Our method is initialized with known positions, and subsequently updates the positions with frame-by-frame tracking.

### How to keep track of separate crazyflies?

Each frame, we acquire a raw point cloud from the motioncapture system. For each object, we use ICP to register the marker positions corresponding to the object's last known pose against the entire scene point cloud. This process is independent for each object, so it can be executed in parallel. This approach assumes that there are no prolonged occlusions during the duration of the flight. We limit ICP to five iterations, which allows operation at 75 Hz with 49 robots using our computer hardware.

### How to mitigate occlusions?

Motion-capture systems sometimes deliver a point cloud with spurious or missing points; if undetected, this may cause tracking errors. To mitigate these errors, we compute linear and angular velocities from the ICP alignments and reject physically implausible values as incorrect alignments. In combination with our on-board state estimation, a few missing frames do not cause significant instabilities, making tracking reliable in practice.

### Initialization algorithm: to allow for small deviations

With identical marker arrangements for each vehicle, the object tracker needs some additional source of information to establish the mapping between vehicle identities (radio addresses) and spatial locations. We currently supply this information with a configuration file containing a fixed initial location for each vehicle. However, it is not feasible to place each vehicle at its exact configured position before every flight. To allow for small deviations, we perform a nearest-neighbor search within a layout-dependent radius about the initial position, and try ICP with many different initial guesses for vehicle yaw. We accept the best guess only if its resulting alignment error is low (less than 1 mm mean squared Euclidean distance between aligned points).

### Communication loss

To reduce communication bandwidth requirements and maintain robustness against temporary communication loss, we fuse motion-capture and IMU measurements onboard in an Extended Kalman Filter (EKF). The filter is driven by IMU measurements at 500 Hz and estimates the states  $(p, v, q)$  where  $p \in \mathbb{R}^3$  is the vehicle's position,  $v \in \mathbb{R}^3$  is its velocity, and  $q \in \mathbb{S}^3$  is the unit quaternion transforming the vehicle's local coordinate frame into world coordinates.

We use two different kinds of communication: request-response and broadcasting. Request-response is primarily used for configuration while the Crazyflie is still on the ground. This includes uploading a trajectory, changing flight parameters such as controller gains, and assigning each Crazyflie to a group. Broadcasting is used during the flight to minimize latency for position feedback, and to achieve synchronized behavior for taking off, landing, starting a trajectory, etc. Broadcast commands can be restricted to subsets of the swarm by including a group ID number.

Our communication does not use a standard transport layer and hence, we need to handle sporadic packet drops as part of our protocol. In order to achieve low latency, we do not aim for guaranteed packet delivery, but rather for a high probability of reliable communication. In our protocol, all commands are idempotent, so they can be received multiple times without any side effects. This allows us to repeat request-response commands until acknowledged or until a timeout occurs. Swarm-coordination commands, such as taking off, do not wait for an acknowledgement but are repeated several times for a high probability that all Crazyflies receive the command. Since external pose estimates are sent at a high fixed rate every 10 ms, there is no need to explicitly repeat such messages.

### Step change in controller setpoint

Commanding a quadcopter to switch from hovering to elliptical motion produces a large step change in the controller setpoint, potentially causing instability. Our system overcomes this issue by using the single-piece polynomial planner to plan a trajectory that smoothly accelerates into the ellipse, iteratively replanning with longer time horizons until it achieves a trajectory that respects the vehicle's dynamic limits. The procedure is general and can be used to plan a smooth start from hover for any periodic trajectory.

### Interactive avoid obstacle mode

The planners discussed so far can follow predefined paths, but are not suitable for dynamically changing environments. For the case of a single, moving obstacle at a known location, such as a human, we use a specialized avoid-obstacle mode. The planner is fully distributed and only needs to know the quadcopter's position  $p$ , its assigned home position  $p_{home}$ , and the obstacle's position  $p_{obst}$ .