# Integration of UAV and Detection

This course includes:

- Single-Threaded approach
- Multi-Threaded approach
- Python Threads
- Thread vs. Process in CPython
- Example of Multi-Threaded approach

# System

- Asynchronous Data Flows
  - Camera
  - Stats
  - Objection detection
  - …
- Dependencies
  - Camera → Objection detection
  - Objection detection → UAV Controlling
  - Stats → UAV Controlling

# Single-Threaded Approach (Polling)

- Implementation
  - Use a single infinite loop to check every data flow.
- Pros
  - Easy to implement.
- Cons
  - High latency tasks can slow down the program.
  - The latency of a data flow is unpredictable.
    - For example, you want to accelerate a UAV and check the speed every few seconds. If the program gets stuck at a certain data flow, the checking frequency would be reduced.

# Multi-Threaded Approach

- How to implement?
  - For every data flow, create a separate thread to handle it.
- Pros
  - Independent latencies of data flows.
  - Flexible modular design.
- Cons
  - Need to handle race condition

# Python Thread Programming

# Thread Example

```
from threading import Thread
import time

class Counter(Thread):
    def __init__(self):
        super().__init__()
        self.A = 0
        self.B = 0

    def run(self):
        start = time.time()
        while time.time() - start < 0.1:
            self.A += 1
            self.B += 1

    def get(self):
        return self.A, self.B

if __name__ == '__main__':
    counter = Counter()   # create a new thread
    counter.start()       # start counter thread
    print(counter.get())  # print counter value
    time.sleep(0.05)      # wait for 0.01 second
    print(counter.get())  # print counter value
    counter.join()        # wait for counter to finish
```

# Output

```
(33796, 33795)
(438475, 438475)
```

# Rcae Condition Example

```python
from threading import Thread
import time

class Counter(Thread):
    def __init__(self):
        super().__init__()
        self.A = 0
        self.B = 0

    def run(self):
        start = time.time()
        while time.time() - start < 0.1:
            self.A += 1
            self.B += 1

    def get(self):
        return self.A, self.B

if __name__ == '__main__':
    counter = Counter()           # Create a counter thread
    adder = Comparison(counter)   # Create a comparison thread
    counter.start()               # Start counter thread
    adder.start()                 # Start comparison thread
    counter.join()                # Wait for counter to finish
    adder.join()                  # Wait for comparison to
                                  # finish
```

```python
class Comparison(Thread):
    def __init__(self, counter):
        super().__init__()
        self.counter = counter

    def run(self):
        start = time.time()
        while time.time() - start < 0.1:
            A, B = self.counter.get()
            if A == B:
                print('%2d == %2d' % (A, B))
            else:
                print('%2d != %2d' % (A, B))
            time.sleep(0.005)
```

## Output

```
30234 == 30234
103092 != 103091
177617 != 177616
252520 != 252519
325453 == 325453
438308 == 438308
513160 == 513160
600583 == 600583
```

# Thread Safe Example

```
from threading import Thread, Lock
import time

class Counter(Thread):
    def __init__(self):
        super().__init__()
        self.A = 0
        self.B = 0
        self.lock = Lock()

    def run(self):
        start = time.time()
        while time.time() - start < 0.1:
            with self.lock:
                self.A += 1
                self.B += 1

    def get(self):
        with self.lock:
            return self.A, self.B

if __name__ == '__main__':
    counter = Counter()           # Create a counter thread
    adder = Comparison(counter)   # Create a comparison thread
    counter.start()               # Start counter thread
    adder.start()                 # Start comparison thread
    counter.join()                # Wait for counter to finish
    adder.join()                  # Wait for comparison to
                                  # finish
```

```
class Comparison(Thread):
    def __init__(self, counter):
        super().__init__()
        self.counter = counter

    def run(self):
        start = time.time()
        while time.time() - start < 0.1:
            A, B = self.counter.get()
            if A == B:
                print('%2d == %2d' % (A, B))
            else:
                print('%2d != %2d' % (A, B))
            time.sleep(0.005)
```

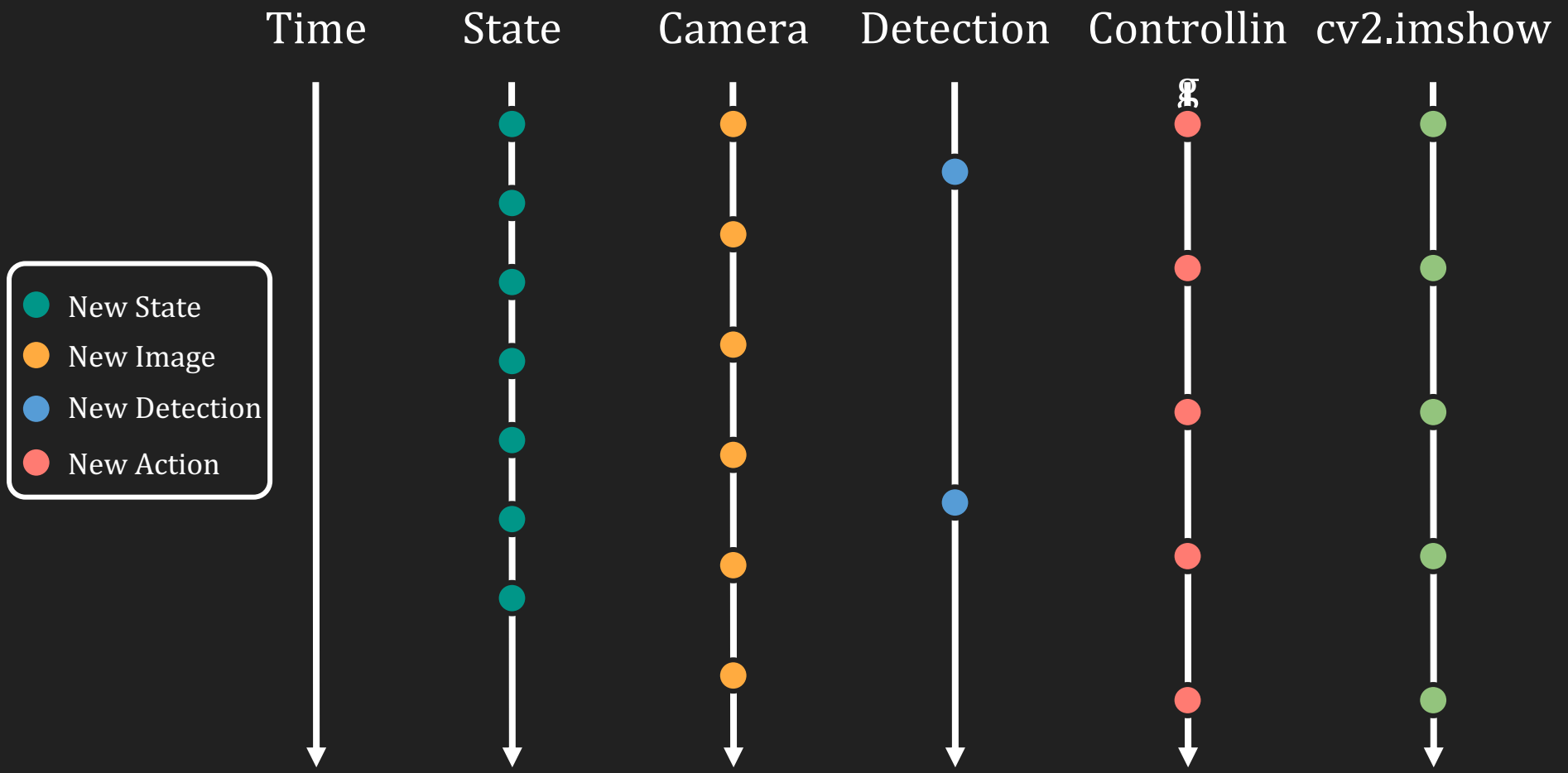
## Output

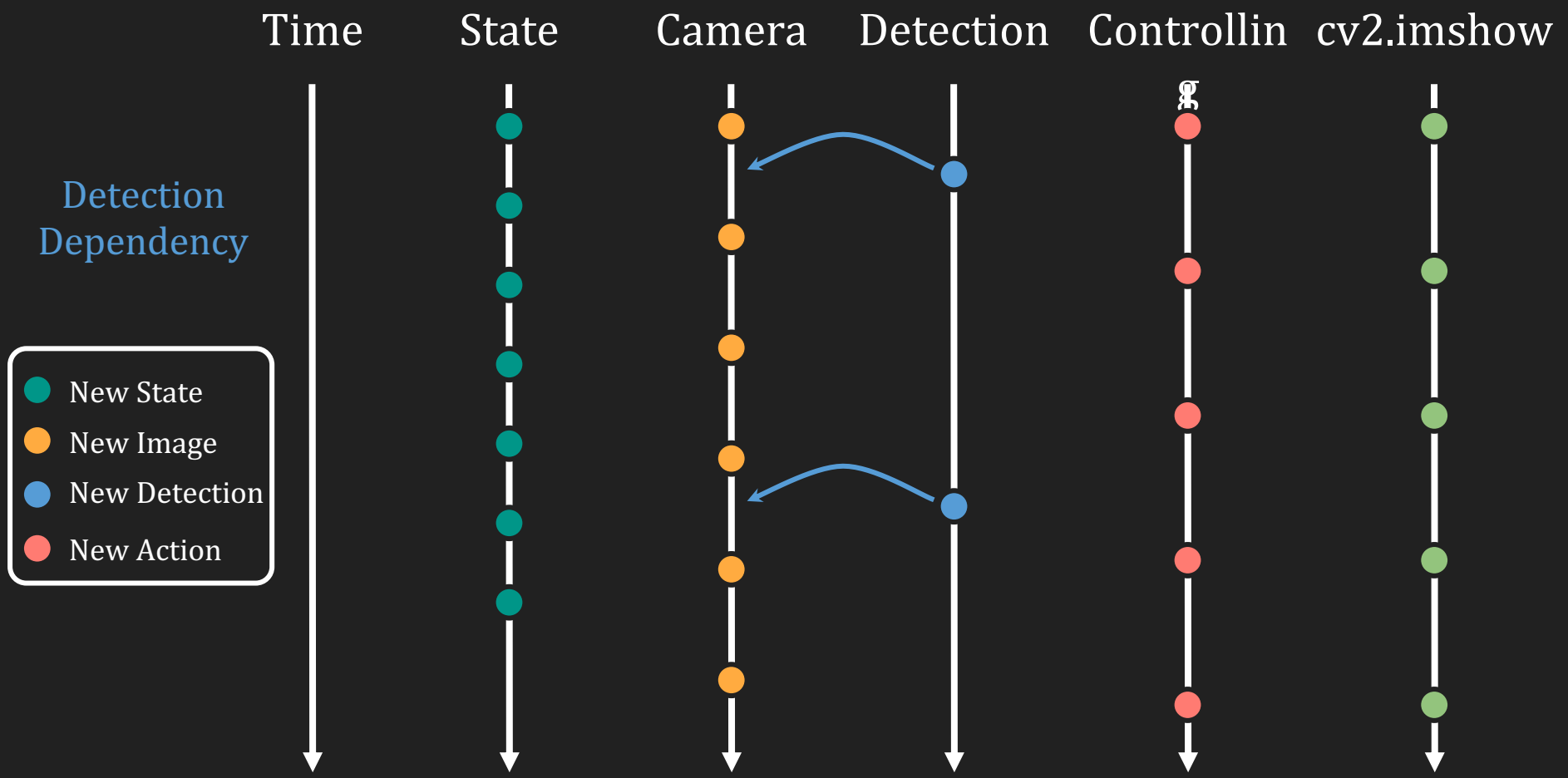
```
61086 == 61086
121860 == 121860
162975 == 162975
223263 == 223263
264737 == 264737
306318 == 306318
329354 == 329354
329354 == 329354
329354 == 329354
```

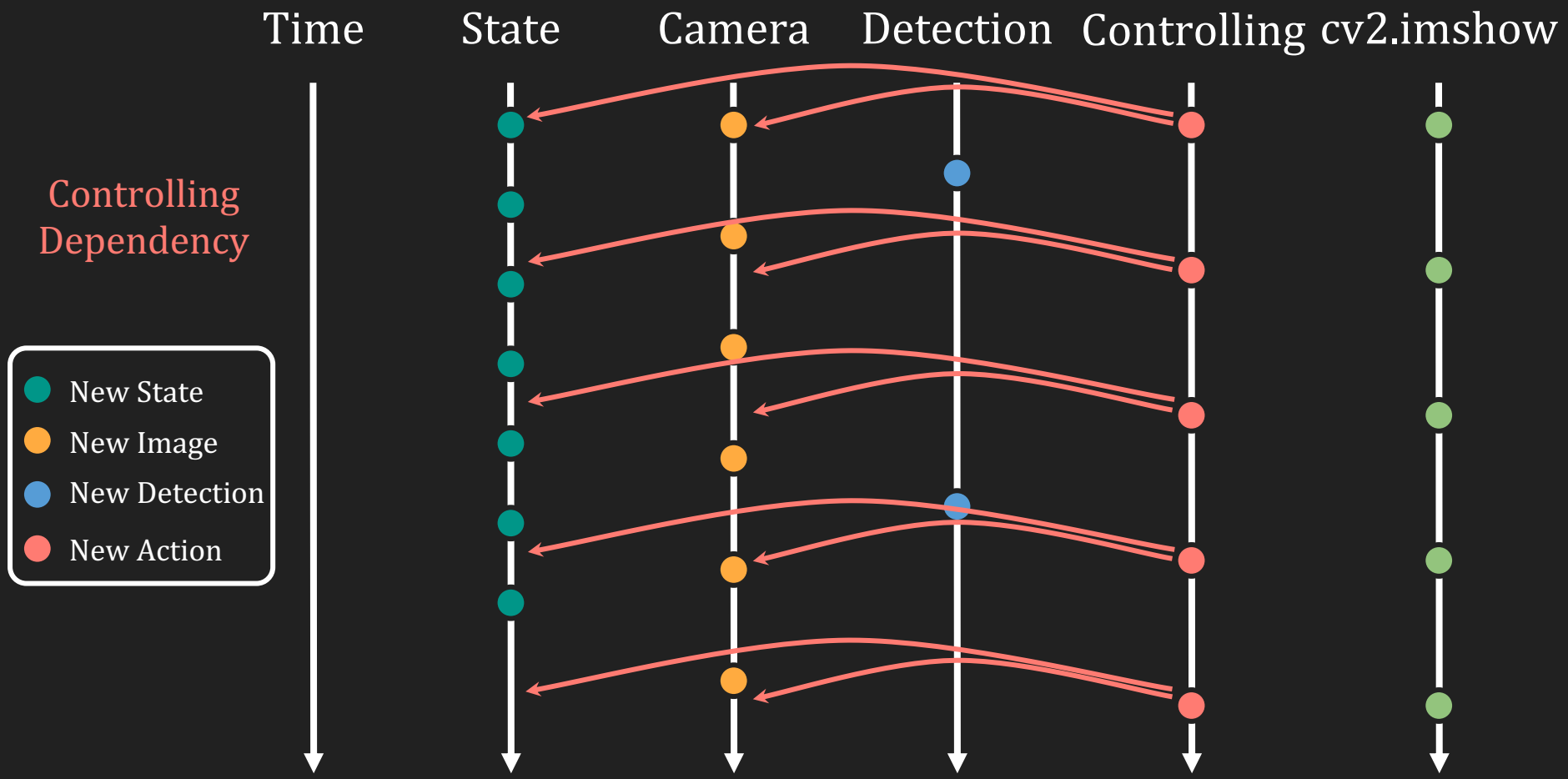
# Thread vs. Process in CPython

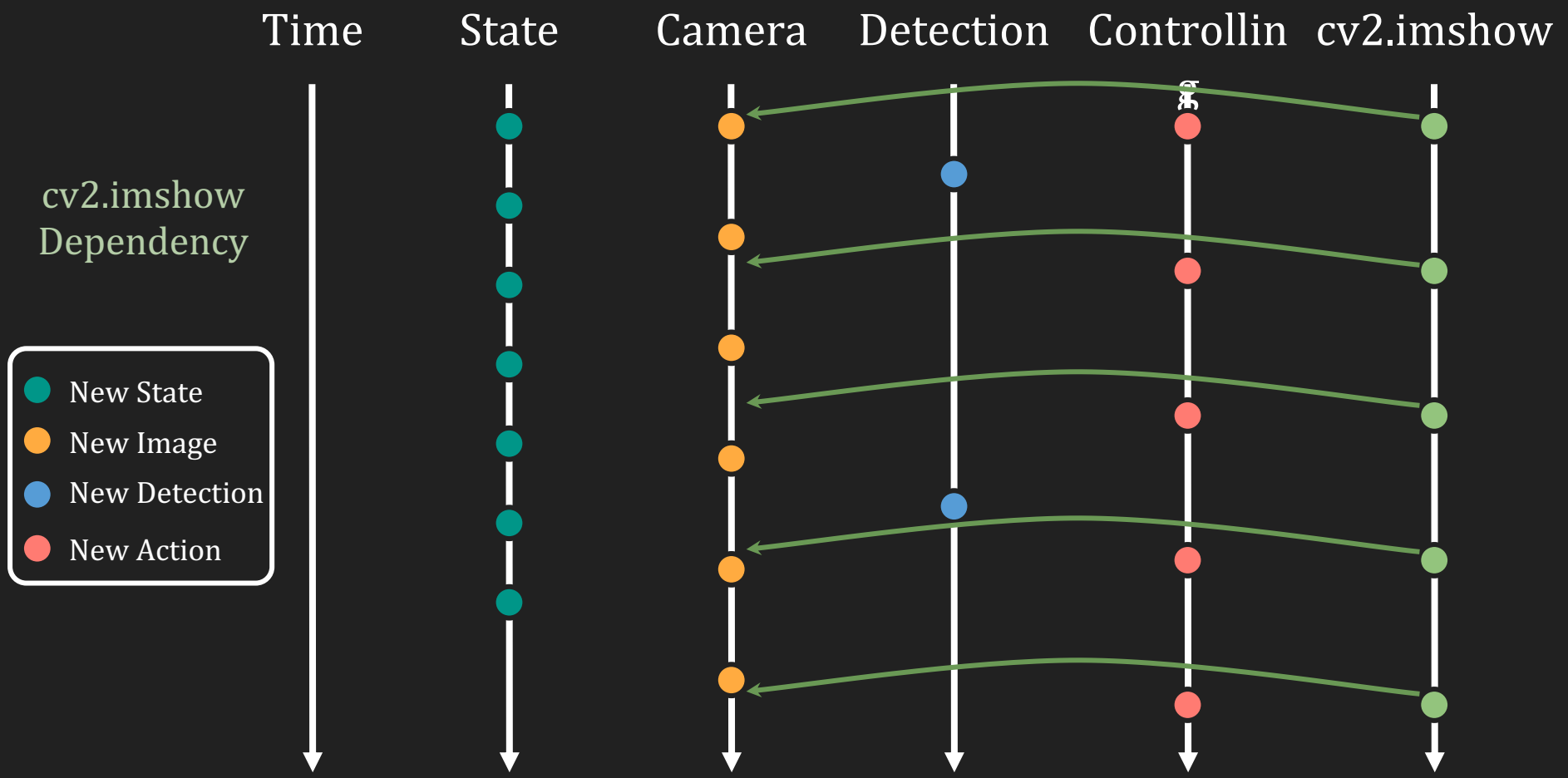
- Threads share memory and are suitable for I/O-bound tasks. However, threads are subject to the Global Interpreter Lock (GIL), which limits parallelism by allowing only one thread to execute Python bytecode at a time.
- Processes in CPython are not subject to the Global Interpreter Lock (GIL), as each process has its own Python interpreter and memory space. Therefore, multiple processes can execute Python code simultaneously on different CPU cores.

# Multi-Threaded Approach


Time
State
Camera
Detection
Controllin
cv2.imshow
New State
New Image
New Detection
New Action

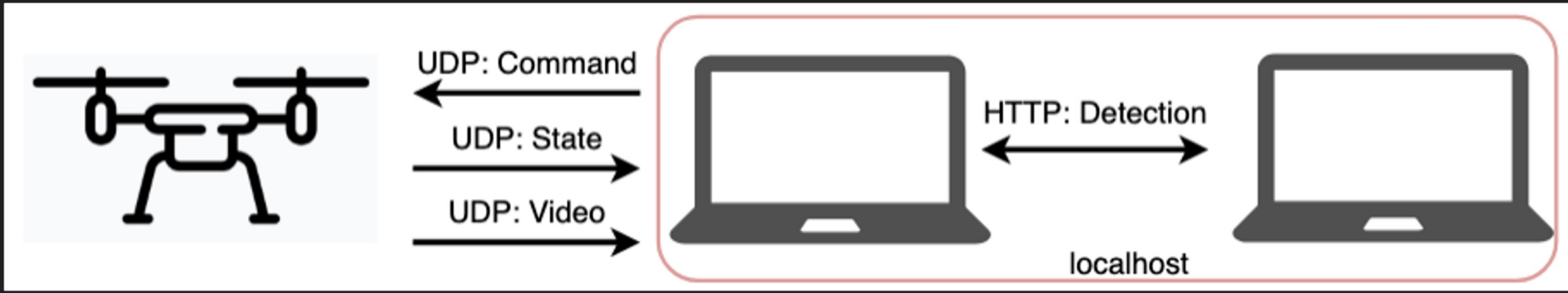

Time
State
Camera
Detection
Controllin
cv2.imshow
Detection Dependency
New State
New Image
New Detection
New Action


Time
State
Camera
Detection
Controlling
cv2.imshow
Controlling Dependency
New State
New Image
New Detection
New Action


Time
State
Camera
Detection
Controllin
cv2.imshow
cv2.imshow
Dependency
New State
New Image
New Detection
New Action

# Multi-Threaded Approach

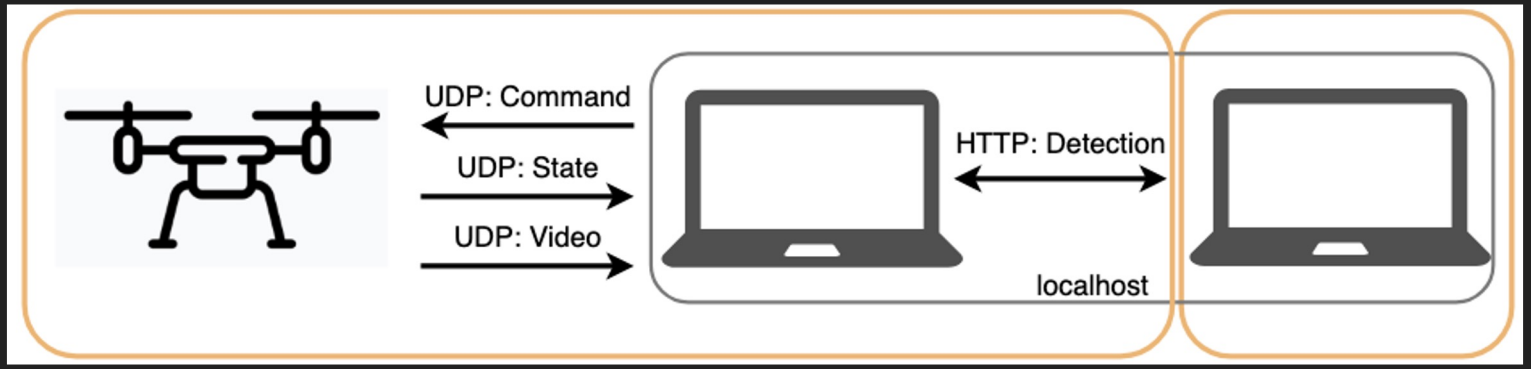
Detect using a local computer



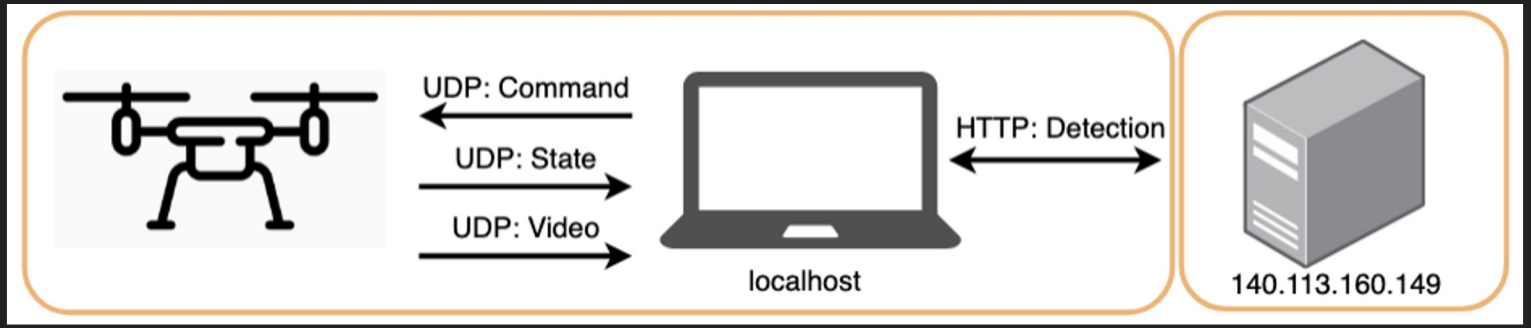

Detect using a remote server

# Multi-Threaded Approach


Detect using a local computer




Detect using a remote server



# Run the Example on Your Laptop

- Clone or pull the source code from github.

```
$ git pull
```

- Install python packages:
  - `lab3/requirements.txt`.
- If you are using a Linux Distros and require CUDA support, don't hesitate to ask TA for help.
- For those using Apple Silicon M series chips, Metal Performance Shaders (MPS) will automatically activate for acceleration.
- If you are using Windows and want to use an Nvidia GPU, you can ask TAs for help or google it.

# Run the Example on Your Laptop

- Start the local detection server using the following command:

```
$ python server.py \
    --port 8888 \
    --weights ./weights/yolov7-w6.pt
```

- For Apple Silicon M series User, use the following command:

```
$ PYTORCH_ENABLE_MPS_FALLBACK=1 python server.py \
    --port 8888 \
    --weights ./weights/yolov7-w6.pt
```

# Run the Example on Your Laptop

- To test your local detection server, open another terminal and use the following command:

```
$ python testserver.py \
    --host 127.0.0.1 \
    --port 8888 \
    --img ./data/street.jpg
```

# Integration of UAV and Detection

- Keep the detection server running.
- Connect to UAV, and start streaming with detection using command:

```
$ python tello.py \
    --host localhost \
    --port 8888
```

# Explanation - A short version

- To add your routing algorithm, simply edit the following code snippet in tello.py.

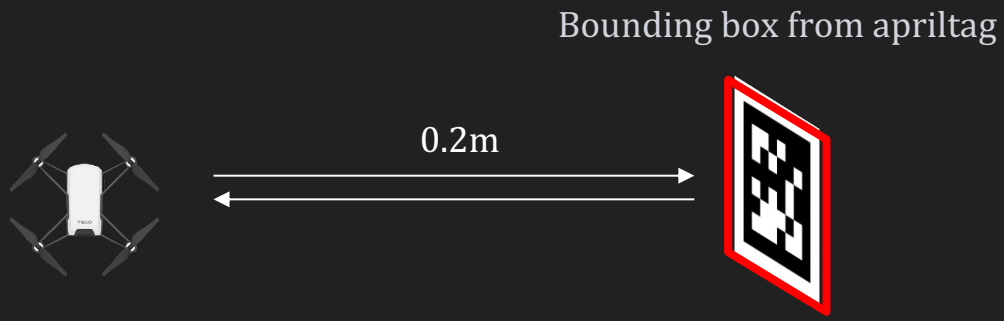
```
    def run(self):
        # res = transmitter.send('takeoff')
        # print('[takeoff]: %s' % res)
        prev_id = None
        while not self.stopped():
            ...
            print('-' * 80)
            print("Battery: %d%%" % state['bat'])
            print("X Speed: %.1f" % state['vgx'])
            print("Y Speed: %.1f" % state['vgy'])
            print("Z Speed: %.1f" % state['vgz'])
            for bbox, score, label, name in zip(bboxes, scores, labels, names):
                # center (x, y) and box size (w, h)
                x, y, w, h = bbox
                ...
            prev_id = id
            # --------------------------
            # Add your routing policy here
            # --------------------------

        # res = transmitter.send('land')

        # print('[land]: %s' % res)
```

# Explanation - A full version

- `StoppableThread` is the ancestor of all threads. Its derived class `ResourceThread`, also known as Receiver, is the ancestor of all receivers. `ResourceThread` has a virtual function `get_result(self)`, which must be implemented by all receivers. This function provides a way for threads to obtain resources and must be implemented in a thread-safe manner. For more information on implementing ResourceThread, please refer to `StateReceiver`, `ImageReceiver`, and `DetectionReceiver`.
- In `tello.py`, `MovingPolicy` inherits from `StoppableThread` and is responsible for reading the drone's status and detection results and giving the drone instructions to move accordingly.
- If you want to construct a new thread to collect different resources, such as collecting detection results for pink boxes, you can inherit from `ResourceThread` and implement `get_result(self)`. Then, in tello.py, add an instance of the newly constructed thread to the threads list, which will be started at the beginning of the program and gracefully terminated when the program is interrupted by ctrl + c.

# Control Policy Tips – Estimate the Position

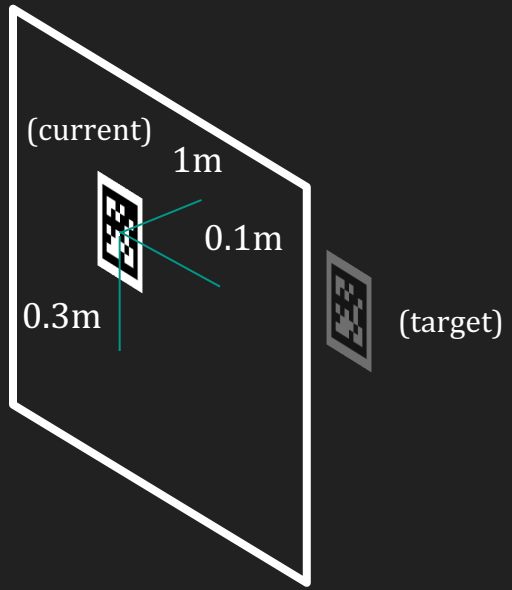
- There are two ways to perform localization
  - Distance to apriltag through estimate the camera pose (hard but stable)
  - Size and position of the bounding boxes (simple but unstable)



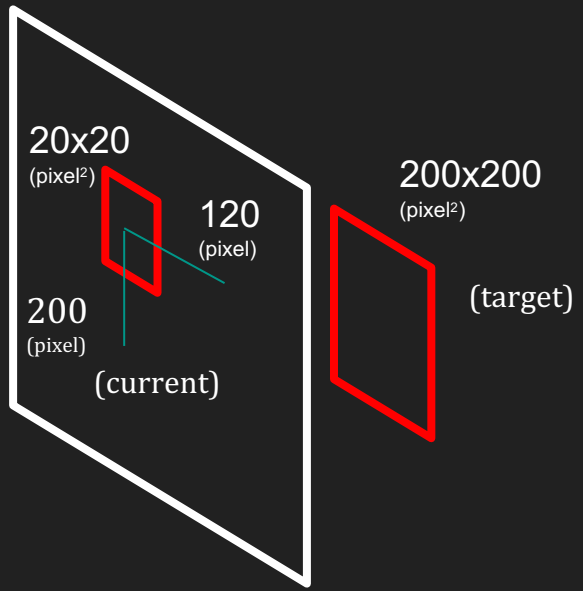
# Control Policy Tips – Estimate the Error

- Set your target and estimate the error

- Camera pose

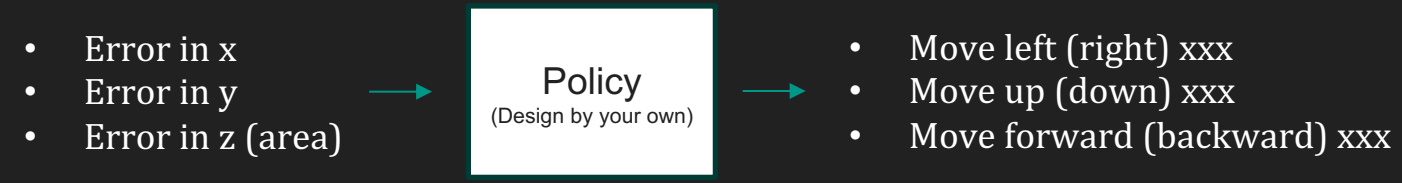



- Bounding Box



# Control Policy Tips – Convert Error to Control

- Setup your policy to convert error to the drone control




If you want to try some advanced methods, PID is an option