AnDrone: Virtual Drone Computing in the Cloud

Alexander Van't Hof
Columbia University
New York, NY
alexvh@cs.columbia.edu

Jason Nieh
Columbia University
New York, NY
nieh@cs.columbia.edu

Abstract

With the continued proliferation of drones, unmanned aerial vehicles, additional uses for them are growing and the demand for their services is on the rise. We present AnDrone, a drone-as-a-service solution that makes drones accessible in the cloud. AnDrone pairs a cloud service with the first drone virtualization architecture. This enables a physical drone to run multiple virtual drones simultaneously in an isolated and secure manner at little additional cost, as computational costs are cheap compared to the operational and energy costs of putting a drone in the air. AnDrone virtualizes drones using a novel Linux container architecture. Android Things virtual drone containers provide a familiar user and development environment that can run existing Android apps. A real-time Linux flight controller container supports existing drone flight software and provides virtual drones with geofenced flight control. A device container transparently multiplexes access from virtual drones to a full range of drone hardware devices, including cameras and other sensors. Upon flight completion, virtual drones and their data can be uploaded to the cloud for offline access. We have implemented an AnDrone prototype based on Raspberry Pi 3 drone hardware. We demonstrate that it incurs minimal runtime performance and energy overhead, supports real-time virtual drone flight control, and runs untrusted third-party software in virtual drones in a secure manner while ensuring that the service provider maintains control of the drone hardware.

CCS Concepts • **Computer systems organization → Embedded and cyber-physical systems**; • **Security and privacy** → *Virtualization and security*; • **Networks** → *Cloud computing*; • **Software and its engineering** → *Virtual machines*; *Operating systems.*

Keywords drones, unmanned aerial vehicles, cloud computing, virtualization, containers, real-time

ACM Reference Format:
Alexander Van't Hof and Jason Nieh. 2019. AnDrone: Virtual Drone Computing in the Cloud. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303969

1 Introduction

Recent advancements in drone technology have allowed the use of drones, unmanned aerial vehicles (UAVs), in applications from aerial photography to package delivery, as well as a wide array of surveying, inspection, and security applications. Smaller, feature limited consumer drones have become more affordable and user-friendly, but still maintain a steep learning curve, requiring significant time investment before becoming proficient in their use and remain prohibitively expensive for infrequent use. Larger and more capable drones remain expensive and out of reach for most consumers, both in terms of cost and complexity. For all drones, users must learn and follow regulations such as where drones can and cannot fly, registration, and licensing, increasing the burden placed on users wanting to use a drone, particularly those with only the occasional use for them. As drone usage continues to rise [66], it is likely that these burdens will only further increase as the need to manage limited airspace only grows. These operational costs coupled with the limited flying time available with most drones due to energy constraints make the time that a drone is actually flying quite valuable. Each flight should be leveraged to its fullest to offset these costs, but despite this, drones today are typically monotasking with a single task assigned for each flight. Given two tasks, there will be two separately owned drones used, each requiring proficient operators and airspace to operate, even if the two tasks would involve the same flight path.

To address these challenges and make drones more widely available, we introduce AnDrone. AnDrone is a drone-as-a-service solution that makes drones accessible in the cloud to interested third parties. With companies like Amazon, UPS, and DHL investigating mass rollouts of delivery drones [4, 27, 65], AnDrone can enable these drones to also be made available to interested third-parties via the cloud to provide additional services. A drone previously tasked with a simple delivery can now simultaneously survey vehicle traffic conditions for a local news company while en route to a delivery, routinely survey a construction site's progress, or photograph a property for a real estate agent. AnDrone enables these use cases without requiring the user to obtain additional hardware or have in-depth knowledge about drones.

AnDrone provides third-party users with lightweight virtual drones that can be configured in the cloud with various apps and services of interest to a user, then safely deployed and multiplexed on real drone hardware. With AnDrone, multiple third-party virtual drones may run simultaneously and continuously throughout a single physical flight as the drone travels from one waypoint to another. At each waypoint, the virtual drone can be given control of the drone and additional device access can be granted allowing the virtual drone to complete any required tasks. AnDrone takes advantage of the observation that computational costs on drones are cheap compared to the operational and energy costs of putting drones in the air, making it is very efficient to multiplex multiple virtual drones on a physical drone to maximize the utility of drone flight time.

AnDrone introduces a novel Linux container architecture to support and isolate different drone execution environments. Unlike, traditional hardware virtualization approaches [13, 15, 16, 24], AnDrone's lightweight container approach [45, 52] pairs well with drone hardware that tends to lack hardware virtualization support and be resource constrained given size, weight, power, and cost considerations. Each virtual drone has its own containerized Android Things environment with which to run its tasks and can provide online interactive access to the drone during flight. AnDrone utilizes Android Things [35] to offer users a familiar and well-known environment with many existing apps, and developers the ability to leverage a large existing base of code, libraries, development tools, and resources, all tailored for Internet of Things (IoT) systems such as drones. To control physical device access to isolate virtual drones from one another and preserve drone hardware safety, AnDrone introduces a device container for managing and multiplexing device access and a real-time Linux flight container for drone flight control.

The device container isolates devices from virtual drones by encapsulating all physical drone devices in a separate isolated execution environment. Only the device container has access to physical devices, allowing it to be used to gate and multiplex access to those devices from virtual drones. Isolating devices in their own execution environment is made possible by leveraging how apps in Android Things interact with devices via higher-level system services [12]. These services allow apps to be transparently decoupled from low-level device implementations and interfaces so that devices can be separated from the rest of the Android Things execution environment. Unlike other container-based hardware multiplexing approaches [6], our approach requires no explicit per-device support, significantly reducing the effort needed to support new platforms and devices. The device container further creates the illusion for the physical devices that each such device is only being used by one task at a time, providing easy compatibility with existing drone-specific software and hardware stacks which are often not designed to support multiplexing. By allowing virtual drones to remain independent of physical devices, they can also be easily moved as needed to different physical hardware.

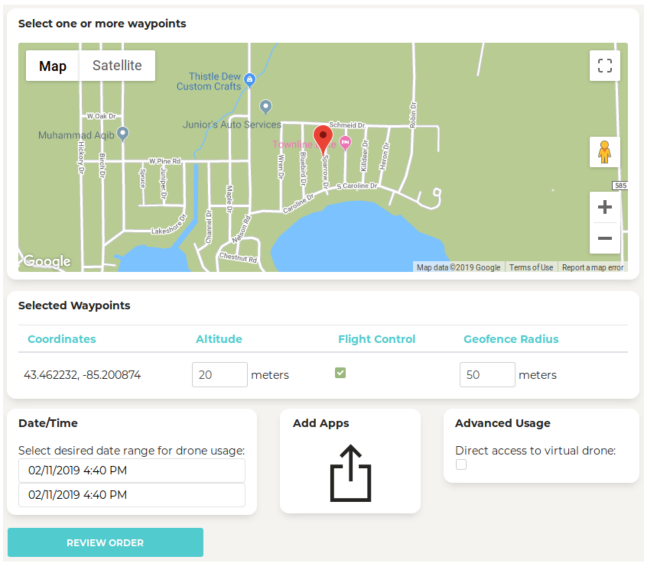
The flight container mirrors the device container approach and isolates the critical real-time flight software stack from virtual drones by encapsulating all flight control logic in a separate isolated execution environment. Only the flight container has access to the physical hardware for flight control, allowing it to be used to gate and multiplex access to flight control from virtual drones. The flight container also allows a different execution environment for the flight software stack, which is crucial for software compatibility as it is based on real-time Linux, not Android. A simple network proxy-based approach enables Android Things virtual drones to interoperate with the real-time Linux flight container.

AnDrone leverages its device and flight containers to provide location-based and conditional drone control. Access to devices such as cameras, camera gimbals, sensors, and GPS can be conditionally granted to virtual drones. Similarly, virtual drones can be geofenced and restricted to operating within a defined set of control parameters. This is used to provide device isolation among virtual drones; a virtual drone restricted to operate in one locale can be prevented from operating in another. This is also used to provide operational safety, for example disallowing overly aggressive maneuvers and enforcing obstacle avoidance. Drone providers can customize the degree of control a user is given over a drone, even restricting it to only operating in a guided mode wherein the drone can only be given destination coordinates and a velocity with which to reach it. With various device restrictions possible, multiple third parties may securely run tasks throughout a single flight and operate a drone in-turn, without interference with each other or the flight stack, fully maximizing the potential of a drone's flight.

We have implemented an AnDrone prototype supporting multiple Android Things virtual drones on drone hardware based on the Raspberry Pi 3 Model B [60] and Emlid Navio2 [30] daughterboard. Our experimental results demonstrate runtime performance overhead of less than 1.5% for a single virtual drone, a negligible effect on drone energy usage, the ability to multiplex multiple virtual drones while ensuring low-latency performance within 300μs of an idle system and sufficient to meet the real-time requirements of drone flight, and that untrusted third-party software may run in virtual drones without undue risk to the physical drone.

2 Usage Model

With AnDrone, users with little to no drone experience can obtain a virtual drone equipped with premade apps to control the drone to accomplish a desired task. For basic drone service, users interface with the AnDrone web portal to order and configure a virtual drone, AnDrone assigns the virtual drone to a physical drone to perform the desired task, then the data from the drone is uploaded back to the AnDrone web portal for the user to access. More advanced options are also available, for example to provide interactive control of the drone during flight.



Select one or more waypoints

Selected Waypoints

Coordinates	Altitude	Flight Control	Geofence Radius
43.462232, -85.200874	20 meters	✓	50 meters

Date/Time

Select desired date range for drone usage:
02/11/2019 4:40 PM
02/11/2019 4:40 PM

Add Apps

Advanced Usage

Direct access to virtual drone: ☐

REVIEW ORDER

Figure 1. Snippet of the AnDrone web portal interface

To order a virtual drone, a user accesses the AnDrone web portal, shown in Figure 1, selects one or more way waypoints, locations where the drone should go, and specifies a desired date and time range for using the drone. A list of possible drone types available is then presented for selection, e.g. drones specializing in obtaining video, drones equipped with specialized sensors, etc. Apps can then be uploaded on to the virtual drone, including by selecting from existing apps available in the AnDrone app store. For example a real estate agent who wants aerial photography of a house can go to the AnDrone app store and find an app that will do this. It could be a basic app that simply circles a geographic location, a more advanced app leveraging computer vision to obtain better results, or even a service-based app offering another company's pilot to manually obtain results. AnDrone leverages Android Things to make it easy for both app developers and users to build on a familiar and established app ecosystem.

Once an app has been selected, the user will use the portal to supply the app with any arguments it requires, e.g., an area on a map to survey. Any further interaction with the drone or app after take off is app-specific. The app may supply a front-end that the user can run on their smartphone or in a web browser to see additional status information or make additional input, or it may act fully autonomously and simply offer the user files it generates. The virtual drone will return flight control once it has completed its task. The user will be informed once the drone has finished its flight, and emailed a link to any files the app generated for them.

If a user requires more advanced functionality, direct access to the virtual drone can be provided instead of just specifying an app to run. When ordering a virtual drone for advanced usage, the user also specifies any devices they need access to and whether they need access to those devices both at and between waypoints or just while operating at a waypoint. If immediate usage is requested of a virtual drone, AnDrone will provide the user with an estimated operating window of when to expect the drone to arrive at the first waypoint so the user can then take over control of the drone. If the user is flexible with regard to when the drone launches, AnDrone will provide an estimated operating window a day in advance of the flight to confirm if it is acceptable to the user. Once the drone takes off, the user is notified via email or text message and the portal provides provides access information for the virtual drone, notably its IP address and port information and how the user may connect to it, much like any recently deployed cloud-based server. The user can then access the virtual drone remotely and run tasks on the virtual drone throughout the entirety of its flight, but flight control is only provided to the user at the specified waypoints of the virtual drone. If flight control is not requested at a waypoint, AnDrone will simply fly the drone on to the next waypoint after arrival. Such waypoints are useful to, e.g. guide a virtual drone along a highway to survey traffic. If flight control is given, the user may return control of the drone at any time via the portal or an app running in their virtual drone. Like other Android systems, device access is provided by Android drone apps. For example, an app running on the drone can forward the camera feed to a client app running on the user's smartphone. All communication between the drone and the user takes place via a cellular internet connection.

It would be unsafe to not enforce restrictions on a virtual drone's control of a physical drone. So once AnDrone hands over control to the virtual drone, the drone is geofenced and restricted to operating within a defined set of control parameters. The extent of these restrictions is flexible and can include multiple aspects of drone flight. For example overly aggressive maneuvers can be disallowed, forced obstacle avoidance can be added, flight modes can be restricted, etc. This allows for a range of functionality varying from allowing users full control of the drone with only basic restrictions on extreme maneuvers, to only allowing the drone to operate in a mode such that it is given destination coordinates and a velocity with which to reach it. With such restrictions the drone can still be pathed wherever the user wants, but always in a predictable manner. The size of the geofence that is applied to the drone is requested by the user up to a maximum size when ordering the drone via the AnDrone portal, with a default size provided.

In addition, AnDrone's containerized design in combination with its device access control ensures privacy and isolation among virtual drones. For example, it is possible that a user A's virtual drone has multiple waypoints and requests access to devices such as the camera while the drone is operating between them. While routing between these waypoints, another user B's virtual drone waypoint may be visited. In such cases, for privacy and conflicting device control reasons, user A's device access will be suspended by default until the drone has finished at user B's waypoint. AnDrone assumes that a user would generally not want another party to have access to the drone's camera or microphone while operating at the user's waypoint.

Like all cloud resources, AnDrone billing is based on usage, but unlike, e.g. a cloud server where time can be used as the billing unit, a drone's flight time is limited and can vary greatly with both the type of drone and how the drone is operated. AnDrone can bill traditional cloud services such as storage or network bandwidth based on regular usage, but bills drone usage based on energy consumption, like a traditional energy utility service. Energy is used for billing drone usage because of its direct correlation with the most critical resource for drones, as well as the ability to leverage the familiarity of energy utility pricing. Estimates of flight time based on energy usage [17, 25, 29, 42, 64] are provided to the users when ordering a drone and the user's specify a maximum billing charge, which in turn specifies the maximum energy the user's drone can consume at their waypoints.

It is possible that the task a user wishes to perform (either via an app or direct access) is unable to be completed on a drone for various reasons, including exceeding the user's maximum billing charge or unpredictable events such as inclement weather. In these cases, virtual drones are instructed to save their current state so that they can be resumed on a later flight.

```
{
  "waypoints": [
    { "latitude": 43.6084298,
      "longitude": -85.8110359,
      "altitude": 15,
      "max-radius": 30
    },
    { "latitude": 43.6076409,
      "longitude": -85.8154457,
      "altitude": 15,
      "max-radius": 20
    }
  ],
  "max-duration": 600,
  "energy-allotted": 45000,
  "continuous-devices": [
  ],
  "waypoint-devices": [
      "camera",
      "flight-control"
  ]
  "apps": [ "com.example.survey.apk" ],
  "app-args": [
    { "com.example.survey": {
        "survey-areas": [
          { "43.6084298,-85.8110359": [
              [43.6087619, -85.8104110],
              [43.6087968, -85.8109877],
              [43.6084570, -85.8110225],
              [43.6084240, -85.8104646]
            ],
          { "43.6076409,-85.8154457": [
...
}
```

Figure 2. Virtual drone definition for example construction site surveys

3 Virtual Drone Definition

To be able to place a virtual drone on a physical flight the following must be known about it: where it is to operate, how much energy it may use, how long it can operate, which devices are needed, when those devices are needed, and what apps should be installed and run. To accomplish this, AnDrone defines a virtual drone as a JSON specification in combination with an Android Things container image. Upon receipt of a new virtual drone JSON specification, AnDrone creates a clean Android Things container and installs any specified apps in it. From then on, the JSON specification and container defines the entirety of the virtual drone. A virtual drone definition is fully self-contained and can be easily reinstated on any drone or even non-drone hardware so long as the CPU architecture matches and the kernel is equipped with Android's kernel features. Each virtual drone container image consists only of its differences from a base virtual drone image, allowing for minimal storage requirements when running multiple virtual drones and storing them offline.

Figure 2 is an example of an AnDrone virtual drone JSON specification, which has seven components. First, the specification has a list of *waypoints* a virtual drone is to visit, each of which is defined by a desired *latitude*, *longitude*, *altitude*, and *max-radius* in meters, which defines a spherical volume from the given waypoint coordinates. Together these parameters define a geofence that will be applied to the virtual drone's control of the real drone, if flight control has been requested. *Max-duration* in seconds and *energy-allotted* in joules combine to specify the maximum time and energy allotted for the virtual drone to operate at all of its waypoints, whichever is exhausted first dictating when control must be taken away. Max-duration is specified in such cases where virtual drones are allowed to land, thus preventing them from idling on the ground indefinitely. *Continuous-devices* specify the list of devices that the virtual drone should have access to continuously once its first waypoint is reached until it completes operation at its last waypoint. *Waypoint-devices* specify the devices that the virtual drone should have access to only while operating at waypoints. Waypoint-devices are prioritized above continuous-devices, so continuous-device access is susceptible to temporary removal should another party's virtual drone's waypoint be visited in between specified waypoints. Flight control can only be specified as a waypoint device, not a continuous device. *Apps* specifies a list apps that should be installed in the virtual drone's container. *App-args* specify the arguments that should be passed to apps when they are started as given by the user when the drone was ordered via the AnDrone portal, in this case being sets of latitude and longitude pairs defining geographic regions that will be surveyed for each waypoint.

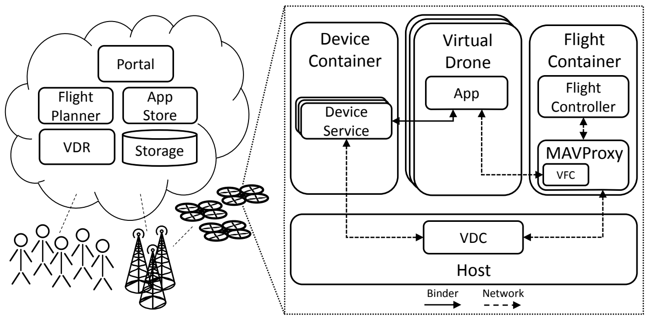


Figure 3. Overview of AnDrone architecture

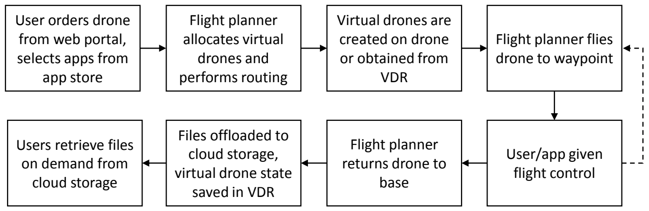

User orders drone from web portal, selects apps from app store
Flight planner allocates virtual drones and performs routing
Virtual drones are created on drone or obtained from VDR
Flight planner flies drone to waypoint
User/app given flight control
Flight planner returns drone to base
Files offloaded to cloud storage, virtual drone state saved in VDR
Users retrieve files on demand from cloud storage

Figure 4. AnDrone workflow

4 AnDrone Architecture

To support virtual drones, AnDrone pairs a cloud service for configuring, allocating, and storing virtual drones offline, with an onboard drone virtualization architecture to safely share physical drone hardware during flight while restricting access to the drone overall to secure it from untrusted third parties. Figure 3 shows the overall architecture of AnDrone. We provide a brief overview of the cloud service, then focus the rest of our discussion on the drone virtualization architecture components and how they interact with the cloud service.

As shown in Figure 3, the cloud service has five components: the AnDrone web portal users use to order their virtual drones, the AnDrone app store that provides apps for virtual drones, general storage for drone flight data, a virtual drone repository (VDR) which stores preconfigured virtual drone definitions for later use or reuse, and a flight planner which allocates virtual drones to physical drone flights and autonomously pilots drones from waypoint to waypoint. AnDrone's flight planner is based on the multirotor drone energy consumption model and the drone delivery routing algorithm developed by Dorling, et al. [29] for assigning deliveries to a fleet of drones to minimize delivery time subject to a drone fleet size constraint. AnDrone assigns virtual drones to physical drones using this model and algorithm by specifying the drone fleet size, using waypoints as delivery locations, and adjusting the energy cost to account for the energy allocated for virtual drones at their waypoints. A limitation of the algorithm is that it treats all waypoints independently, so users may not prescribe that waypoints be traversed in a specified order and the algorithm may decide to visit waypoints of one virtual drone in the middle of a set of waypoints of another virtual drone. Providing a planner algorithm that can support waypoint ordering and grouping is an area of future work. During flight, the cloud service communicates with drones over cellular internet as current LTE performance is already sufficient for cellular based drone control [58]; future cellular technology is being developed with mission critical drone usage in mind [57]. Figure 4 shows the workflow of an AnDrone flight and where each cloud service component is involved.

The onboard drone virtualization architecture is built on a Linux operating system (OS), given that drones are primarily ARM-based and Linux is the dominant OS for ARM devices. To enable running multiple virtual drones alongside a real-time Linux-based flight stack, AnDrone's virtualization architecture uses Linux containers to support running multiple variants of Linux at the same time, including Android Things. AnDrone containerizes all Linux instances to provide isolation among them and manage their resources as necessary to ensure the reliability and performance of all containers. To provide real-time support for containers, AnDrone's Linux kernel is augmented with the PREEMPT_RT [47] patches to make it fully preemptible to minimize latencies for real-time tasks. Remote access to containers is provided by tunneling all communication over a per-container virtual private network (VPN), allowing potentially insecure protocols, such as those used by drone flight controllers, not originally intended for use across the Internet to now be used securely over cellular internet communication.

By relying on containers instead of traditional hardware virtualization [19, 22–24, 46], AnDrone removes the need to emulate numerous sensor devices, potentially introducing unacceptable latencies, and expands hardware compatibility to devices without hardware virtualization support. This is essential since due to the size, weight, power, and cost considerations drone hardware tends to be resource constrained and lacking the virtualization capabilities familiar to server hardware where virtual machines are commonly used. Additionally, AnDrone is able to maximize the limited resources of drone hardware by avoiding the need to run multiple full OS instances. By leveraging Android Things, an Android variant specifically designed for resource constrained IoT devices, AnDrone offers app developers a well-known environment and off-the-shelf reuse of Android apps and code inside a minimal, more resource-efficient OS than stock Android. Additionally, with out-of-the-box support for single board computers like the Raspberry Pi, Android Things offers better hardware support than stock Android for devices commonly used in drones.

As shown in Figure 3, the virtualization architecture has four main components: the virtual drone containers loaded onto the drone hardware, a device container for multiplexing device access, a flight container for virtualizing and multiplexing flight control, and a virtual drone controller (VDC) that manages virtual drones. We discuss each of these components in further detail below.

4.1 Virtual Drone Containers

Each virtual drone container appears to applications as an independent Android Things instance which is isolated from other virtual drone instances. For efficiency on the drone, virtual drone containers are managed using Docker [28] so that each container consists of common read-only base disk images layered together with a writable layer on top [54–56]. Common read-only base disk images can be shared across virtual drones, making virtual drones easier to manage and reducing storage costs. Docker also simplifies management by providing built in commands that enable AnDrone to easily move virtual drones back to the cloud and to other drone hardware as well as store them offline in the cloud. In addition, Docker enables AnDrone to prevent abuse and excessive consumption of resources, which can interfere with other virtual drones by allowing AnDrone to place restrictions on the resources each virtual drone can use.

While Docker is useful for supporting multiple Android Things virtual drones, it is not sufficient. Unlike traditional desktop and server computing environments, Android, and platforms that it runs on such as smartphones and IoT systems, incorporate a plethora of devices that applications expect to use. Some devices can be easily virtualized because they need not provide much of the original device functionality. For example, Android cannot be run without a graphical user interface and expects to be able to access a framebuffer device. Since drones are headless, the framebuffer contents are not actually displayed. In this case, each container can be simply given a virtual framebuffer device to use rather than the real one, and the virtual framebuffer device can just be a memory region in which contents can be written. No actual hardware device support is needed. However, for more complex devices that actually need to provide full featured functionality, existing approaches provide no effective mechanism to enable apps to directly leverage these device features from within virtualized environments, whether they be traditional virtual machines (VMs) or Docker containers. These devices are instead intended to be used directly by a single Android instance and cannot be used by multiple instances simultaneously, which AnDrone requires for supporting multiple simultaneously running virtual drones.

One key Android device is not actually a hardware device, but a software abstraction, namely Android's Binder interprocess communication (IPC) device. Binder is a software abstraction that functions as Android's primary IPC mechanism and is utilized by processes via various ioctl system calls. Binder inherently provides isolation as no communication can occur between a client and a service without first obtaining a handle to that service; services exist as *nodes* that clients reference via an integer-based per-process *handle*. To obtain a handle to a node, a client must be given it by the node itself or someone who already has a handle to that node. In Android, services register themselves with the userspace

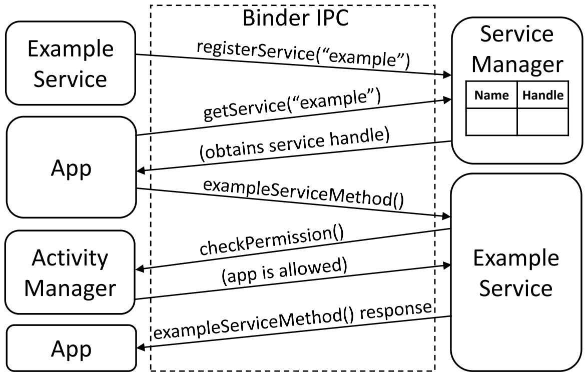


Figure 5. Android Binder service communication

ServiceManager, Binder's *Context Manager*, which itself is always obtainable through Binder via the handle 0. The ServiceManager retains a mapping of handles to corresponding names of services given at registration time. Apps can obtain handles to desired services by requesting a reference from the ServiceManager, as shown in Figure 5. Binder only allows one Context Manager, which offers handles to all services. However, AnDrone runs multiple virtual drone containers, each an Android Things instance, and each expecting to be have their own Context Manager offering their own services.

To achieve this, we add device namespaces [5, 6, 21] to Binder to isolate the Context Manager to a per-container level, allowing each virtual drone instance to have its own Context Manager. When a ServiceManager registers as a Context Manager, Binder identifies the container from which the ServiceManager registers so that subsequent references to the container's handle 0 will reference the respective container's own ServiceManager instead of one global one. Since Binder does not allow access to services without access to their respective handles, the end result is that each container's clients and services are isolated from those in other containers.

While device namespaces are useful for enabling the Binder device to operate in the context of virtual drone containers, this is not as useful for actual hardware devices that need to deliver full functionality, in some cases involving complex and proprietary device drivers. Augmenting these complex implementations with device namespaces would be problematic both due to complexity and lack of availability of source code. While a virtual device could be introduced, it would still need to provide access to the actual hardware device functionality and could not simply be a dummy virtual device, which comes around to the original question of how to multiplex the hardware device among virtual instances.

4.2 Device Container

Unlike traditional desktop and server systems, Android IoT systems are highly vertically integrated in which several layers of software are involved on a given system to offer a tall interface from apps to hardware devices. Apps are written in

Service	Device(s)
AudioFlinger	Microphone, Speakers
CameraService	Camera
LocationManagerService	GPS
SensorService	Motion, Environmental Sensors

Table 1. Listing of device container services

Java and call Java frameworks, which function as libraries that provide the core public APIs used by developers for Android functionality including accessing devices. Frameworks use Java Native Interface (JNI) to package up calls and pass them through Android's Binder IPC mechanism to communicate with Android system services, which are system processes that run in the background and are used to manage devices. Apps do not interact with hardware devices directly, but instead via system services.

Ideally, these device services can be used to multiplex hardware for multiple containers as they are already designed to multiplex access to hardware devices from multiple processes. In a vanilla Android instance, system services would run as part of the Android instance. However, with multiple Android instances, this cannot be done as running multiple system services, each directly accessing devices, would cause conflicts. Alternatively, running multiple system services would require an additional mechanism injected below system services in the middle of a complex device stack to somehow multiplex their access to hardware devices, which would be a difficult challenge.

To solve this problem, AnDrone introduces a *device container*, a special container running a minimal Android instance with direct access to hardware devices to run Android's device services. Only a single set of system services are run, just like a vanilla Android instance, and AnDrone leverages the multiplexing functionality in system services to support access to system services and their underlying devices from multiple virtual drone containers. System services are centralized in the device container and removed from all virtual drone containers. AnDrone then makes these services available in all virtual drone containers in place of their own.

To allow virtual drones to use the device services running in the device container, those services need to be registered with each virtual drone's ServiceManager so that the respective ServiceManager can provide a reference to the desired device service when requested by an app. To support this cross-container service registration, we add a new ioctl to the Binder driver, *PUBLISH_TO_ALL_NS*, callable only by the device container for security. Figure 6 shows how the ioctl is used. When the device container's ServiceManager receives a new service registration request, it checks to see if the service name is in a pre-specified list of services that are to be shared as shown in Table 1. If the service name is in the list, the ServiceManager calls this ioctl to publish it in all running virtual drone containers. The ioctl then takes the service name

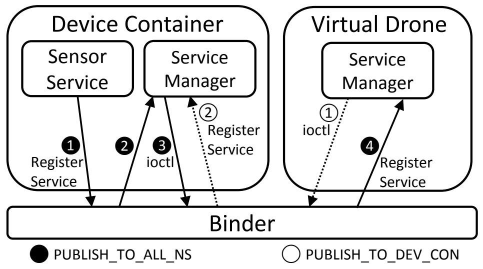


Figure 6. Device container service publishing process

and handle passed to it and checks all other containers for existing ServiceManagers. The presence of a ServiceManager indicates that the container is a virtual drone running Android Things. The ioctl then makes its own registration call to these existing ServiceManagers with the provided name and handle, thus registering the device container's service inside the virtual drone container. The same process will be performed in the future for any newly created virtual drone containers. AnDrone also disables the equivalent device services inside the virtual drone containers from starting by modifying init files and Android's *SystemServer*, a process responsible for starting many services. When an app in any virtual drone container asks for a reference to one of the shared services, the respective ServiceManager will return a reference to the single service running inside the device container. Once the app has the reference to the shared service, it can communicate with it via Binder the same as if it were in its own container. All communication with the device services listed in Table 1 is fully encapsulated in Binder messages or by using a file descriptor shared via a Binder message.

Although device services are now available in virtual drone containers, a service must also allow apps to use it. In Android, a service asks the ActivityManager if the calling app has permission to use it. This means that a device service will ask the device container's ActivityManager for permission rather than the calling container's ActivityManager, which will be problematic because the device container's ActivityManager will not be aware of the permissions for apps running outside of the device container. To address this problem, we add one more new ioctl to the Binder driver, *PUBLISH_TO_DEV_CON*. We modify each container's ServiceManager to call this ioctl when its respective ActivityManager registers itself with the ServiceManager, as shown in Figure 6. The ioctl appends the ActivityManager service name with the container identifier and registers it with the device container's ServiceManager. We then modify Android's native and Java *checkPermission()* functions in the device container to request the calling container's ActivityManager from the device container's ServiceManager, as identified by the modified service name. To allow services to identify the calling container so that it can perform this check, we make a small modification to Binder to include the calling process' container identifier in its transaction data structure alongside the existing calling process' PID and EUID.

AnDrone's device container model introduces minor additional risk by sharing several services among all of the containers, breaking isolation for those specific services. We deem this an acceptable trade-off as these services are already meant to be used by untrusted apps and thus are already hardened. Billions of active Android devices help demonstrate how secure these already are. Note that our approach does not add any security risk beyond existing Android configurations which share device services among apps. Compared to a standard Android environment with dozens of shared services, AnDrone's shared services represent an order of magnitude reduction in attack surface via system service exploitation. Even if a vulnerability were found in one of the shared device services, it does not fully compromise virtual drones or the whole of the device container and would depend on the device service vulnerability. In the worst case, if the flight controller, discussed in Section 4.3, is running on shared hardware with the virtual drones and the GPS or SensorService are compromised, stability and control of the flight can be compromised by the attacker. However, these two services are relatively simple compared to other system services and to the best of our knowledge, and after a review of Android's Common Vulnerabilities and Exposures (CVEs), have never had significant security vulnerabilities discovered in them. Section 4.3 discusses how this additional risk to flight control can be mitigated.

4.3 Flight Container

A drone's flight is controlled via a flight controller. The flight controller is commonly a native Linux daemon running on the drone itself that is responsible for both stabilizing the drone and accepting commands for maneuvering it. Communication with the flight controller commonly takes place via the Micro Air Vehicle Link (MAVLink) protocol, allowing a ground station or app to have full control of the drone via any underlying medium. To support AnDrone's usage model, we must be able to multiplex the flight controller among virtual drones as well as between virtual drones and the cloud-based flight planner.

To address this problem. AnDrone introduces a flight container for running the flight controller in its own standard Linux container, isolating and prioritizing it over virtual drones due to its mission critical importance. We leverage and modify MAVProxy [7], a portable, minimalist ground control station with MAVLink proxying capabilities, to allow multiple clients to connect to the flight controller. MAVProxy acts as an intermediary between clients and the flight controller, which provides an indirection mechanism to virtualize the flight controller. AnDrone uses MAVProxy to give the cloud-based flight planner full native access to the flight controller, but presents each virtual drone with its own virtual flight controller (VFC) to control the degree of flight control allowed. MAVProxy provides a standard unrestricted flight controller connection for the flight planner and service provider to use, and a VFC connection for each virtual drone which restricts the flight control commands that will be accepted and presents a virtualized view of the drone that differs from that of the physical drone. The extent of the restricted commands is configurable via a whitelist of MAVLink commands available as a number of preconfigured whitelist templates which are customizable by the service provider. The most restrictive template available will only allow the drone to operate in guided mode wherein only a desired GPS position may be given. The least restrictive template allows for full control of the drone so long as it remains within the geofence.

A virtual drone can connect to its VFC at anytime throughout a flight, but until a virtual drone's waypoint is reached, the VFC presents a view of their drone as idle on the ground at the waypoint to indicate it is inactive and declines any commands sent to it. As the real drone approaches a waypoint, the virtual drone presented automatically takes off to meet the physical drone's position. Once the real drone's position is met, the virtual flight controller begins to accept commands. The commands sent to it from this point on will control the physical drone, but the drone is both geofenced and the commands that the VFC will accept are restricted. The exception to this virtualized view is if the virtual drone has continuous access to devices while operating between its waypoints. To prevent a discrepancy between the view of the drone and device readings, the actual drone's position is given, but commands are still declined until a waypoint is reached. Once the virtual drone is finished with flight control, or is forced to finish, the VFC again refuses to accept commands and presents the drone as landing, where it stays for the remainder of the flight. Meanwhile the physical drone is piloted on to the next waypoint.

MAVLink and flight controllers already support containing drones with geofences, but the action taken when the geofence is breached is to perform a failsafe landing. For AnDrone, this behavior is undesired as the flight must continue so that other virtual drones may operate and eventually return to base. We augment the geofence support such that a breach causes the following steps to be performed: inform the virtual drone of the breach, disable commands on the VFC connection, guide the drone back inside the geofence, and switch it into loiter mode to hold its current position. Flight control is then returned to the virtual drone. With this approach, geofence breaches can be safely handled without interruption to the overall flight.

To run the flight container on the same hardware as the virtual drones, the flight controller must also have access to hardware devices, such as the GPS, which are controlled by and must be accessed via the device container, just like any other virtual drone. However, the device container provides Android-based service interfaces, which are not supported by native Linux. AnDrone introduces additional hardware abstraction layer (HAL) support to the flight container to provide a Binder based bridge between the controller and the device container's device services. Adding HAL support for sensor devices, e.g. barometer, is straightforward since sensor access is supported via the Android Native Development Kit

(NDK). However, the NDK does not provide access to GPS, so a native interface for Android's LocationManagerService had to be created.

Although the flight stack is isolated from virtual drones by containerization, it is still vulnerable to kernel-level faults and vulnerabilities. When sharing hardware with the flight controller, a bug or intentional kernel crash can result in loss of control of the drone. This potential risk is not unique to a shared environment and the risk of a kernel crash is typically handled through additional failsafe mechanisms. For example, the Emlid Navio2 daughterboard includes a failsafe in the on-board microcontroller [68]. This risk can be removed by running the flight controller on separate hardware if desired.

4.4 Virtual Drone Controller

To manage virtual drones, AnDrone provides a Virtual Drone Controller (VDC). The VDC is a daemon running natively on the host OS of the physical drone responsible for managing virtual drone containers. Prior to each flight, the flight planner sends the VDC the virtual drone definitions assigned to it. The VDC creates containers for each virtual drone to run as, or if resuming a previous virtual drone flight, obtains the existing virtual drone from the VDR. Once a flight is complete, if a virtual drone is unable to complete its task prior to exhausting its allotted energy or must be interrupted due to unpredictable reasons such as inclement weather, the VDC is responsible for storing the virtual drone, including its updated container image, in the VDR at the end of a flight so that it may be resumed on a later flight. Although checkpoint-based migration is likely feasible for virtual drones [39, 44, 51], AnDrone simply leverages the existing Android activity lifecycle to facilitate saving and restoring the state of virtual drones so they can be migrated between physical drones. Android apps are informed when they are about to be terminated and allowed to save their current state via the *onSaveInstanceState()* callback. The apps can then use this saved state when starting once again to restore themselves as they were prior to being terminated. All AnDrone apps are expected to support this standard Android functionality. A virtual drone's state can then safely be saved offline as part of its disk image.

The VDC also manages virtual drone device access by verifying whether or not a virtual drone is allowed access to a device throughout a flight. This is done by extending Android's service permission model so that the *checkPermission()* function called when a device service queries the ActivityManager, as discussed in Section 4.2, also queries the VDC. The VDC informs the device service if the calling virtual drone container has permission to use the requested device, as defined by its virtual drone definition. The flight planner notifies the VDC throughout the flight once virtual drone waypoints are reached so the VDC can update its device access restrictions. Unlike Android's service permission model which only checks permissions when an app first asks to use a device then allows the app to retain the permissions, AnDrone must be able to revoke permissions of an app that is actively accessing a device, e.g. when leaving a waypoint. To avoid substantial changes to device services to support permission revocation, AnDrone provides this functionality by asking apps to voluntarily disable device access. As discussed in Section 5, AnDrone apps make use of an AnDrone SDK and are expected to disable device access upon being informed that they are no longer accessible via the AnDrone SDK. Since apps may choose to ignore the permission revocation notification, the VDC enforces this by asking each device service if there are any processes from the given virtual drone still accessing a device after notification, in which case the VDC terminates those processes. In a similar manner, the VDC is queried by the flight container to determine if a virtual drone has permission to control the flight.

5 AnDrone Apps

AnDrone apps are standard Android apps that are written just like any other Android app. However, AnDrone apps also require the ability to interact with AnDrone to know about events specific to AnDrone, such as when they have arrived at a waypoint and when they are finished at a waypoint. AnDrone provides this functionality with a simple AnDrone SDK that apps can use. Figure 7 lists the AnDrone SDK methods.

A key component of this SDK is the *WaypointListener* callback class, as shown in Figure 8. Apps create an instance of this class and register it with the AnDrone SDK method *registerWaypointListener()* listed in Figure 7. Once registered, the app can be notified of various AnDrone related events. An app is notified upon arriving at a waypoint via the *waypointActive()* callback. After receiving this callback, the app knows it is now at the given waypoint, has access to flight control and other waypoint-specific devices it requested, and is free to perform its desired task. Upon leaving a waypoint, either voluntarily or because the maximum time or energy allocation allowed for the virtual drone has been reached, an app is notified via the *waypointInactive()* callback, indicating flight control and waypoint-specific device access is about to be removed and the drone is moving on. The *WaypointListener* also provides callbacks informing apps if their virtual drone is running low on its allotted time or energy allocation via *lowEnergyWarning()* and *lowTimeWarning()*. If the geofence is breached, the app is informed via the *geofenceBreached()* callback, and the app is informed when the virtual drone regains control of the physical drone by a subsequent *waypointActive()* callback. *suspendContinuousDevices()* is called when approaching another party's virtual drone waypoint, indicating that access to devices must be suspended until the other party is finished at their waypoint, as indicated by a call to *resumeContinuousDevices(). waypointCompleted()*, listed in Figure 7, is called by an app to indicate it has finished its task at a waypoint.

Figure 7 lists four additional methods for AnDrone apps. *getFlightControllerIP()* is used to facilitate connecting to the

```
void registerWaypointListener(WaypointListener l);
void waypointCompleted();
InetAddress getFlightControllerIP();
void markFileForUser(String path);
int getAllottedEnergyLeft();
int getAllottedTimeLeft();
```

Figure 7. AnDrone SDK methods

```
abstract class WaypointListener {
     waypointActive(Waypoint waypoint);
     waypointInactive(Waypoint waypoint);
     lowEnergyWarning(int remaining);
     lowTimeWarning(int remaining);
     geofenceBreached();
     suspendContinuousDevices();
     resumeContinuousDevices();
}
```

Figure 8. Simplified WaypointListener class definition

virtual flight controller. *markFileForUser()* is used to indicate files that should be made available to the user in cloud storage after the flight. The final two *getAllotted* functions allow the app to obtain the remaining energy and time allotted for the virtual drone. For advanced end users, who may not be using an app, AnDrone's SDK functionality is also made available to them via a command line utility.

In addition to using the AnDrone SDK, every AnDrone app must include an XML manifest file, similar to the existing Android XML manifest file, indicating the requested device permissions and any arguments it expects from users. The AnDrone manifest is used by the AnDrone portal and flight planner to provide information needed as part of ordering a virtual drone and flight planning. The AnDrone portal reads app arguments from the AnDrone manifest so it knows what arguments an app requires from the user when ordering the virtual drone and prompts the user for these values as part of the ordering process. The AnDrone flight planner needs to know which devices are needed by apps in a virtual drone so it can avoid device access conflicts among virtual drones and control access to devices during flight. Device permission requests are declared in the AnDrone manifest much like existing Android permissions in the Android manifest. A `<uses-permission>` tag is used to specify the `name` and `type` of access requested. `type` can be either that of *waypoint* for devices that only need to be accessed at task waypoints, or *continuous* for access to devices while also between waypoints. Arguments the app requires from the user are declared with an `<argument>` tag, specifying a `name`, `type` of argument, and if the argument is `required`.

6 Evaluation

We have implemented an AnDrone prototype and evaluated it in the context of Linux-based drone quadcopter hardware shown in Figure 9. The quadcopter uses a DJI Flame Wheel F450 Air Frame [20], equipped with four T-Motor MN2213 950Kv motors [62] with 9.5" propellers attached, and four SimonK 30A electronic speed control units to control the speed of the motors mounted beneath the frame. The drone is controlled by a Raspberry Pi 3 Model B [60] (Broadcom BCM2837, 4x Cortex-A53 1.2 GHz CPU, 1GB RAM) single-board computer (SBC) with an attached Emlid Navio2 [30] daughterboard drone controller, Raspberry Pi Camera Module v2 [59], and a SanDisk Extreme 16GB microSDHC card for storage. The SBC is mounted in the center of the Air Frame and the entire drone is powered by a Turnigy 5000mAh 3S battery [38] mounted underneath the SBC. The SBC runs Raspbian [61] Stretch, the official Linux distribution for Raspberry Pi, as the host OS, Android Things v1.0.3 in the virtual drone and device containers, and Alpine Linux [3] v3.7 in the flight container supporting the ArduPilot Copter [9] v3.4.4 flight controller.

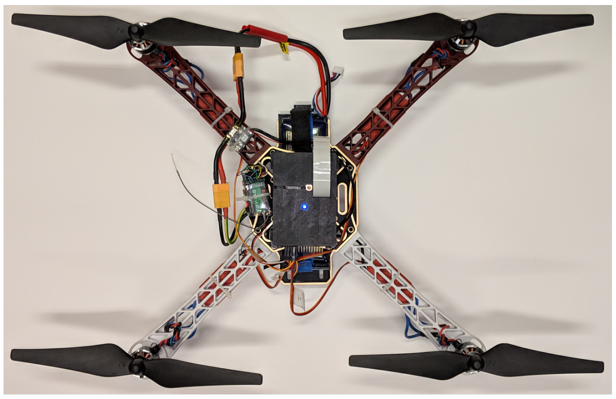


Figure 9. AnDrone quadcopter prototype

6.1 Runtime Overhead

We first evaluate the performance of AnDrone when running multiple virtual drones with various workloads. The first workload we used was the popular Android PassMark PerformanceTest benchmark [53], which is commonly used to measure multi-threaded CPU, disk, and memory performance of Android systems. PassMark also has 2D and 3D graphics benchmarks, but we did not run those as Android Things does not have hardware accelerated GPU support. To measure how runtime overhead is affected by the number of virtual drones running, we ran PassMark in each virtual drone simultaneously with different numbers of virtual drones running. Docker container resource controls were not used. To show the impact of different levels of kernel preemptibility, we measured performance for both the AnDrone default kernel configuration with PREEMPT_RT support enabled versus the minimally accepted real-time support used by Navio2's default kernel configuration with only PREEMPT support enabled. PREEMPT_RT allows the kernel to be almost fully preemptible while PREEMPT disallows kernel preempt when local interrupts are disabled, the latter potentially incurring higher latencies. We normalized PassMark performance compared to running a single instance of PassMark using stock

Android Things natively on the system without AnDrone, which does not have PREEMPT_RT or PREEMPT enabled. Running multiple PassMark instances simultaneously is only made possible through virtual drones so only a comparison with a single instance on stock can be made.

Figure 10 shows PassMark results normalized to the performance of stock Android Things running a single PassMark instance; lower is better. Results are shown for running with one, two, or three virtual drones in total; three virtual drones means that all three were simultaneously running the individual PassMark tests. Three virtual drones running simultaneously was the maximum our prototype could support due to memory constraints. We expect future, more powerful drones will be able to support more virtual drones. PREEMPT_RT results are indicated by the "-RT" postfix, while the other results are for just enabling PREEMPT. With a single virtual drone running, CPU, disk, and memory performance remained relatively constant with at most 1.5% overhead, demonstrating the minimal performance overhead of virtual drones. CPU performance shows roughly a linear decrease in performance with a linear increase in the number virtual drones running PassMark, indicating that runtime overhead does not increase significantly with more virtual drones. With three virtual drones, the PREEMPT_RT kernel performed somewhat worse than the PREEMPT kernel, indicating some cost associated with greater kernel preemptibility and more tasks running. On the other hand, disk and memory performance did not decrease as much with an increase in the number of virtual drones running PassMark. Disk performance overhead with three virtual drones was roughly 2x and 2.2x for the PREEMPT and PREEMPT_RT kernels, respectively. Memory performance overhead with three virtual drones was roughly 1.8x and 2.3x for the PREEMPT and PREEMPT_RT kernels, respectively. In practice, we expect that more realistic apps will experience less performance slowdowns as they benefit from multiplexing more variable resource demands.

6.2 Real-time Latency

To demonstrate that AnDrone can provide the flight controller, ArduPilot, with sufficient real-time latency guarantees in the presence of various workloads, we ran the commonly used latency benchmark, cyclictest [63], and configured it to run in the flight container in the same way as AnDrone runs ArduPilot by locking all memory allocations and assigning its thread the highest real-time priority. We ran three different workloads at the same time as cyclictest and configured cyclictest to run for 100 million loops to provide sufficient samples to have a high confidence in encountering worst case latencies. First, we ran cyclictest on an otherwise idle system to measure baseline performance. Second, to cause latencies an AnDrone environment is likely to encounter under heavy load, we ran cyclictest with three virtual drones running, one idle, one running PassMark continuously in a loop, and one continuously running the iperf [37] network throughput test to stress the system and generate interrupts. Docker container resource controls were not used. Finally, to generate an even worse case latency scenario, we ran cyclictest while stressing all aspects of the system with the stress [69] workload generator to strain CPU, memory, I/O, and disk subsystems, and iperf to strain the network subsystem, both running natively on the host. For both iperf scenarios, iperf was connected over Gigabit Ethernet via a network switch to a Lenovo Thinkpad T540p acting as the iperf server. Stress was configured to run with four CPU worker processes, two I/O worker processes, two memory worker processes, and two disk worker processes. We performed all cyclictests on both the PREEMPT and PREEMPT_RT enabled kernels to compare their performance.

Figure 11 shows the latency of each cyclictest measurement for these three workloads and two kernel configurations. PREEMPT_RT results are indicated by the "-RT" postfix, while the other results are for just enabling PREEMPT. The PREEMPT idle, PassMark, and stress scenarios exhibited maximum latencies of 1,307μs, 14,513μs, and 17,819μs and average latencies of 17μs, 44μs, and 162μs, respectively. The PREEMPT_RT idle, PassMark, and stress scenarios exhibited maximum latencies of 103μs, 382μs, and 340μs and average latencies of 10μs, 12μs, and 16μs, respectively. ArduPilot's most demanding real-time requirement is its most frequently run control loop, the *fast loop*. The fast loop processes values from one or more inertial motion units (IMUs) and adjusts the motors to maintain stability and aid in flying the drone. Ardupilot's fast loop runs at 400Hz, requiring real-time latencies below 2500μs to achieve this. The PREEMPT_RT patched kernel demonstrated latencies well within the requirements of ArduPilot, whereas the PREEMPT kernel did occasionally fall short. However, occasionally missing ArduPilot's fast loop deadline will not cause significant stability issues [11]. Given this, and the infrequency with which the PREEMPT kernel failed to meet ArduPilot's requirements, it is likely this kernel configuration is also sufficient for AnDrone.

To further demonstrate that the stability of the drone is not compromised with AnDrone, we operated our drone prototype at a hover and compared its performance while running the idle and PassMark scenarios described above. The AnDrone default PREEMPT_RT kernel was used for these flight tests. We then analyzed logs of each flight using DroneKit's Log Analyzer [2] and compared them using the Attitude Estimate Divergence (AED) analyzer. The AED analyzer evaluates the flight logs and determines if the flight controller's estimated attitude of the drone differs significantly from the canonical drone attitude, indicating instability if the drone's yaw, pitch, or roll diverges more than 5° from the estimates for longer than .5 seconds. Both scenarios were within normal divergence.

6.3 Memory Usage

Since available memory is the primary limitation on how many virtual drones can be run, we quantified the amount of

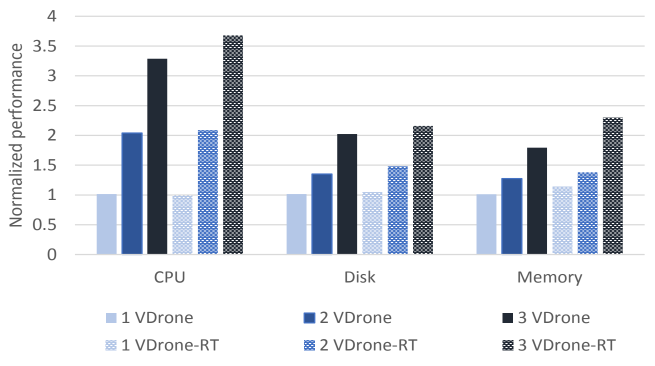


Figure 10. Runtime overhead

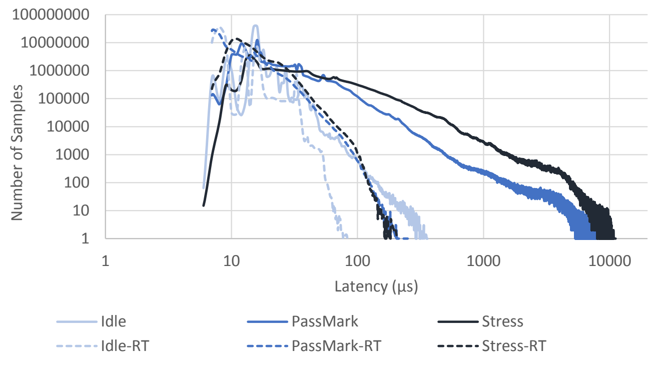


Figure 11. Real-time latency

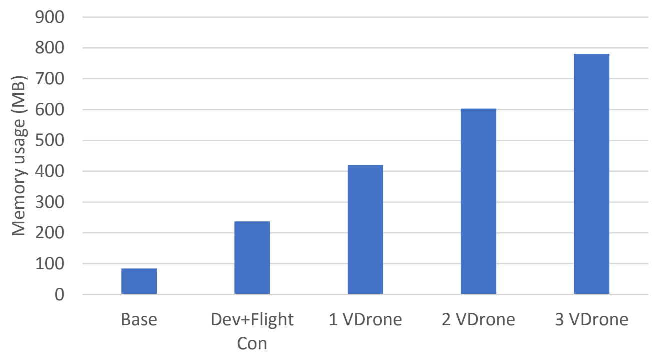


Figure 12. Memory usage

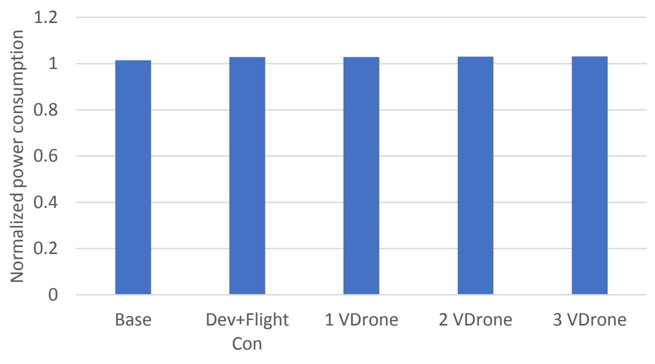


Figure 13. Power consumption

memory used by virtual drones. We first measured the memory usage of AnDrone without any containers, then adding just the device and flight containers, then starting up from one to three virtual drones, the maximum supported by our drone hardware prototype; starting a fourth virtual drone fails due to lack of memory but does not interfere with other virtual drones already running. Each virtual drone was idling on its app launcher screen. Figure 12 shows the memory usage of AnDrone in these various configurations. The results show that less than 100MB of RAM is needed to run the VDC and host OS, roughly 150MB of additional RAM is needed to run both the device and flight containers in addition to the base system, and approximately 185MB is needed for each virtual drone. Although our prototype does have 1GB of RAM, only 880MB is made available after accounting for peripheral I/O reserved space and RAM allocated to the GPU for camera functionality.

6.4 Power Consumption

To demonstrate that AnDrone has a negligible effect on energy usage, we used a Monsoon Power Monitor [50] to measure the power consumption of AnDrone with the drone at rest, normalized to stock Android Things running on the Raspberry Pi idling on its app launcher screen. We measured energy usage using the same system configurations as described in Section 6.3 for measuring memory usage. Figure 13 shows the energy usage of AnDrone in these various configurations, with all configurations within 3% of stock Android Things. In absolute numbers, while at idle with three virtual drones running, AnDrone consumed approximately 1.7W. We also measured the energy usage when fully stressing the system using the same stress and iperf workloads used for measuring real-time latency as discussed in Section 6.2, but the energy usage was the same, 3.4W, across both stock Android Things and all AnDrone configurations, so they are omitted from Figure 13. Both idle and fully stressed system energy usage are insignificant when compared to the power draw of the rest of the drone. Even consumer-level drone batteries are rated to allow a power draw of well over 100W throughout a 20 minute flight.

6.5 Network Performance

Extensive testing and trials by Qualcomm [58] have demonstrated the feasibility of leveraging LTE for real-time control of drones beyond visual line of sight. In a field trial consisting of approximately 1,000 test flights in addition to complementary simulations, Qualcomm demonstrated LTE's ability to support safe drone operation in real-world environments up to 400 feet above the ground with strong signal availability at high altitudes, successful handover and lower frequency of handovers, and comparable coverage to mobile devices on the ground.

To verify their findings with AnDrone and evaluate control of a drone over a cellular network, we used USB tethering to connect our prototype to a Nexus 5X smartphone operating on the T-Mobile cellular network, then conducted various experiments to measure the impact of the cellular network on drone control. First, we qualitatively compared the control responsiveness of flying the drone via a traditional RF-based remote controller versus using the cellular network. For the latter, we connected a Microsoft Xbox 360 gamepad to a Lenovo Thinkpad T540p laptop running the APM Planner 2 [8] ground station and accessing the Internet via the Columbia University campus WiFi network. We did not notice any significant difference in control responsiveness when using the gamepad to operate the drone over the cellular network versus the RF-based remote controller. To quantify the difference, we set up a testbed environment with the prototype disconnected from drone hardware, but still operating on the cellular network. We then issued roughly 150,000 MAVLink commands over a 12 hour period to the flight controller via the Thinkpad T540p laptop using a wired Verizon Fios Gigabit Connection to the Internet. Although the commands did not succeed since the flight controller was not connected to drone hardware, we could measure the latency between when each command was sent and when the flight controller received it. On average, commands took 70ms to be received with a maximum latency of 356ms and a standard deviation of 7.2ms. 6 packets were lost overall. By comparison, the average RF remote control latency of typical hobby drones ranges from 8ms and 85ms [40, 41].

6.6 Multi-waypoint Flight Simulation

To demonstrate AnDrone as a whole under more extensive flight conditions, we used the ArduPilot Software in the Loop (SITL) Simulator [10] to have the AnDrone flight planner deploy virtual drones to various waypoints under simulated flight conditions. For testing with the SITL simulator, we replaced AnDrone's onboard flight controller with a Lenovo Thinkpad T540p laptop running its own ArduPilot flight controller using the SITL simulator software instead of real drone hardware. The virtual drones still communicate with the flight container's MAVProxy via their VFC connections, but MAVProxy communicates with the laptop's simulated flight controller instead of the flight controller inside the flight container. Similarly, the flight planner communicates with the laptop as well. With this setup, we are able to capture core aspects of AnDrone, such as managing device and flight control access among virtual drones, but are not able to capture the effects virtual drones may have on an onboard flight controller on a real-world flight.

We used this SITL simulator setup to perform an AnDrone flight with three virtual drones, one running an autonomous survey app, another running an interactive app allowing remote control of the drone from a Nexus 5X smartphone, and a third providing direct user access. Upon starting the flight, the VDC created and started the three separate virtual drones from their respective virtual drone definitions. Remote console access was provided for the direct access virtual drone at the start of the flight. The flight planner correctly pathed the drone to the first waypoint for the autonomous survey app virtual drone. Once at the waypoint GPS, camera, and flight control was given to the app, which in turn used the DroneKit [1] API to fly back and forth over a location while recording video as an app might do for surveying a field. After calling the AnDrone *waypointCompleted()* SDK method, the flight planner pathed the drone to the second waypoint for the interactive app virtual drone. The app successfully allowed maneuvering the drone, and an intentional geofence breach was handled as expected. At the next waypoint, control was handed to the direct access virtual drone. The APM Planner base station running on the ThinkPad, as discussed in Section 6.5, was used to successfully connect to the virtual drone's VFC. Access to the camera was allowed while at the waypoint whereas previous access attempts were denied. Finally, the drone returned to its base.

7 Related Work

Various approaches have been proposed for providing drone services, though few have been implemented and none of them support virtual drones. An IBM patent application [36] describes an autonomous drone service system to allow users to order specific drone services that will be performed either autonomously or by a ground-based pilot. Unlike AnDrone, users cannot perform any service beyond those offered by the service provider and are not given flight control or access to a drone. Additionally, such a service requires both a drone and a pilot for each order, increasing the cost of such services.

UAV as a Service (UAVaaS) [70] is a proposed framework enabling users to connect to a cloud-based service to use a drone. Unlike AnDrone, UAVaaS does not give direct access to drones, but instead supplies users with cloud-based APIs allowing for control of a drone and access to its device data. Users connect to a cloud-based UAVaaS Coordinator service, which in turn connects to a drone. Because of this, all existing drone and device code is incompatible and new apps and services must be written explicitly for UAVaaS. Additionally, as users interact with the drone through a cloud intermediary and cannot directly run anything on the drone, applications, such as autonomous control apps, may not have sufficient reliability, latency, and bandwidth guarantees to function. For example, if faced with intermittent networking issues, an autonomous AnDrone app will remain unaffected while it conducts its task, whereas any remotely running app will likely have to abort the flight.

Dronemap Planner [43] provides access to drones for developers through web services. To achieve this, a cloud-based MAVLink to WebSocket proxy is created allowing web-based control over a drone's flight. Other than MAVLink proxying, no additional drone functionality is offered and no other device access is possible. Unlike AnDrone, drones can only be navigated and exclusive, unrestricted control is always given to the operator. Any intermittent networking issues will cause the same problems that occur for UAVaaS.

UAV-Cloud [48, 49] provides middleware that facilitates developing collaborative drone apps. Various drone resources such as sensors are made available as cloud-based RESTful APIs for collaborative drone apps to leverage. Beyond defining these APIs, little is actually implemented. Unlike UAV-Cloud, AnDrone's goal is not to abstract away aspects of a drone for collaborative apps, but to offer a complete drone-as-a-service solution making drones accessible in the cloud.

Fly4SmartCity [18, 31] is an emergency-management service that offers aerial support to people in need. A user, e.g. a citizen, requiring aerial support may request it via a mobile app. A cloud service then dispatches a drone and privileged users, e.g. police officers, are given web-based access to its camera feed. Unlike AnDrone, Fly4SmartCity is limited to offering users web-based camera access and is primarily focused on path planning of city-based drones.

FarmBeats [67] is an IoT platform for agriculture that enables data collection from sensors, cameras, and drones to facilitate farm analytics. As part of this, FarmBeats leverages drones connected via an IoT base station, which is in turn connected to a farmer's internet connection via a radio link. These drones map fields, monitor crop canopy, and check for anomalies. Unlike AnDrone, the primary focus of FarmBeats is leveraging user-owned drones, and other sensors, to supply analytical data to the cloud.

Similar to AnDrone's goal of fully leveraging flight time, Galois, Inc. has investigated leveraging the spare capacity of flight controller hardware by porting FreeRTOS to run in a Xen VM on ARM Cortex A15 based devices [26, 32, 33] so the VM can be used to run SMACCMPilot [34]. Their approach suffers from the difficulty of exposing the vast array of sensor devices available to the VM. There has been no indication that full support for running SMACCMPilot inside a Xen VM has been achieved. More generally, there exists ongoing projects for adding support to both the Xen [13] and KVM [14] for running VMs with real-time requirements. This could potentially allow for stronger isolation between a flight controller and third-party VMs. However, challenges for such a system exist given both the lack of hardware virtualization support with drone hardware and the difficulty in supporting and passing low-latency access to the numerous and diverse sensor devices to VMs for use by real-time apps.

Like AnDrone, Cells [6] leverages containers to support multiple Android instances on the same hardware, though Cells focuses on smartphones and tablets where graphics support is of key importance. Cells introduces a new type of namespace for devices to accomplish this, but assumes a more simplified use case model appropriate for smartphones and tablets. With Cells, the user interacts with one foreground Android instance at a time, while background Android instances mostly do not need hardware device access. In contrast, AnDrone must allow for any virtual drone requiring access to physical hardware devices at any time, so more fine grained access control is needed than what is supported by Cells. Device namespace support can be tedious and error prone as new devices may require kernel driver modifications. Because device namespaces require contextual knowledge of how a device operates to implement support for a given device, supporting opaque peripheral devices without specific kernel drivers can be a challenge. These devices maintain their context in userspace, communicate through userspace over buses like Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C), and the kernel only sees raw reads and writes. In contrast, AnDrone's device container design has no such limitation as it operates at the system service level, and requires no per-device support, significantly reducing the effort required to support multiple Android instances on new platforms.

8 Conclusions

We have designed, implemented, and evaluated AnDrone, the first drone-as-a-service solution making drones fully accessible in the cloud. AnDrone introduces virtual drones for the first time, enabling drone tasks for multiple users to be consolidated on the same physical drone and performed during the same flight. Users can create and configure Android Things virtual drones in the cloud with various apps and services of interest, leveraging a large existing base of Android apps, developers, and resources, which are then safely deployed and multiplexed on real drone hardware. AnDrone achieves this by pairing a cloud service with a lightweight virtualization architecture that introduces a new device and flight container design for multiplexing drone hardware and managing virtual drone device access, including providing virtual drone geofenced flight control. Multiple variants of Linux can be run at the same time, including Android Things virtual drones together with a real-time Linux flight controller.

We have implemented an AnDrone prototype and used it together with drone quadcopter hardware based on the Raspberry Pi 3 Model B and Emlid Navio2 daughterboard. Our experimental results demonstrate runtime performance overhead of less than 1.5% for a single virtual drone, the ability to run multiple virtual drones for the first time with performance that scales linearly with workload without any significant increase in energy costs, and sufficient low-latency performance for real-time flight controllers while multiplexing multiple virtual drones. Real-world and simulator-based flight demonstrations show that virtual drones can run simultaneously without compromising the stability and safety of the drone.

9 Acknowledgments

Phillip Stanley-Marbell provided helpful comments on earlier drafts of this paper. This work was supported in part by NSF grants CNS-1717801 and CNS-1563555.

References

[1] 3D Robotics. 2015. DroneKit by 3D Robotics. http://dronekit.io/.

[2] 3D Robotics. 2015. DroneKit Log Analyzer Documentation. http://la.dronekit.io/.

[3] Alpine Linux development team. 2019. Alpine Linux. https://alpinelinux.org/.

[4] Amazon.com Inc. 2016. Amazon Prime Air. https://www.amazon.com/Amazon-Prime-Air/b?node=8037720011.

[5] Jeremy Andrus. 2015. *Multi-Persona Mobile Computing*. Ph.D. Dissertation. Columbia University.

[6] Jeremy Andrus, Christoffer Dall, Alex Van't Hof, Oren Laadan, and Jason Nieh. 2011. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP 2011)*. Cascais, Portugal, 173–187.

[7] ArduPilot. 2017. MAVProxy. https://ardupilot.github.io/MAVProxy/html/index.html.

[8] ArduPilot. 2019. APM Planner 2 Home. http://ardupilot.org/planner2/.

[9] ArduPilot. 2019. ArduPilot Open Source Autopilot. http://ardupilot.org/.

[10] ArduPilot. 2019. SITL Simulator (Software in the Loop). http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[11] Miguel Arroyo, Hidenori Kobayashi, Simha Sethumadhavan, and Junfeng Yang. 2017. FIRED: Frequent Inertial Resets with Diversification for Emerging Commodity Cyber-Physical Systems. *ArXiv e-prints* (Feb. 2017). arXiv:1702.06595

[12] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. 2016. POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys 2016)*. London, UK, 19:1–17.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*. New York, NY, 164–177.

[14] Paolo Bonzini. 2015. Realtime KVM. https://lwn.net/Articles/656807/.

[15] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. 2012. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (Nov. 2012), 12:1–12:51.

[16] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. 2017. *Hardware and Software Support for Virtualization*. Morgan and Claypool Publishers.

[17] Alessandra Cappiello, Ismail Chabini, Edward K. Nam, Alessandro Lue, and Maya Abou Zeid. 2002. A Statistical Model of Vehicle Emissions and Fuel Consumption. In *Proceedings of the IEEE 5th International Conference on Intelligent Transportation Systems*. Singapore, 801–809.

[18] Marcello Chiaberge, Antonio Toma, Gabriele Ermacora, Stefano Rosa, Basilio Bona, Mario Silvagni, Marco Gaspardone, and Roberto Antonini. 2014. *A Cloud Based Service for Management and Planning of Autonomous UAV Missions in Smart City Scenarios*. Vol. 8906. 20–26. https://doi.org/10.1007/978-3-319-13823-7

[19] Christopher Dall. 2018. *The Design, Implementation, and Evaluation of the Linux ARM Hypervisor*. Ph.D. Dissertation. Columbia University.

[20] Da-Jiang Innovations. 2018. Flame Wheel ARF Kit. https://www.dji.com/flame-wheel-arf.

[21] Christoffer Dall, Jeremy Andrus, Alex Van't Hof, Oren Laadan, and Jason Nieh. 2012. The Design, Implementation, and Evaluation of Cells: A Virtual Mobile Smartphone Architecture. *ACM Transactions on Computer Systems (TOCS)* 30, 3 (Aug. 2012), 9:1–31.

[22] Christoffer Dall, Shih-Wei Li, Jintack Lim, Jason Nieh, and Georgios Koloventzos. 2016. ARM Virtualization: Performance and Architectural Implications. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA 2016)*. Seoul, Korea, 304–316.

[23] Christoffer Dall, Shih-Wei Li, and Jason Nieh. 2017. Optimizing the Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 2017)*. Santa Clara, CA, 221–234.

[24] Christoffer Dall and Jason Nieh. 2014. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2014)*. Salt Lake City, UT, 333–347.

[25] Raffaello D'Andrea. 2014. Guest Editorial Can Drones Deliver? *IEEE Transactions on Automation Science and Engineering* 11, 3 (July 2014), 647–648.

[26] Jonathan Daugherty. 2015. Getting Started with FreeRTOS for Xen on ARM. https://blog.xenproject.org/2015/02/02/getting-started-with-freertos-for-xen-on-arm-2/.

[27] Deutsche Post AG. 2014. Unmanned Aerial Vehicles in Logistics. http://www.dhl.com/content/dam/downloads/g0/about_us/logistics_insights/dhl_trend_report_uav.pdf.

[28] Docker Inc. 2019. Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com.

[29] Kevin Dorling, Jordan Heinrichs, Geoffrey G. Messier, and Sebastion Magierowski. 2017. Vehicle Routing Problems for Drone Delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 47, 1 (Jan. 2017), 70–85.

[30] Emlid Ltd. 2017. Emlid Navio2 - Raspberry Pi Autopilot HAT Powered by ArduPilot & ROS. https://emlid.com/navio/.

[31] Gabriele Ermacora, Stefano Rosa, and Antonio Toma. 2016. Fly4SmartCity: A Cloud Robotics Service for Smart City Applications. *Journal of Ambient Intelligence and Smart Environments* 8, 3 (2016), 347–358.

[32] Galois Inc. 2015. Galois Releases FreeRTOS Port for Xen on ARM Systems. https://galois.com/blog/2015/02/freertos-xen/.

[33] Galois Inc. 2016. Update: FreeRTOS for Xen on ARM Systems. https://galois.com/blog/2016/07/update-freertos-xen-arm-systems/.

[34] Galois Inc. 2017. SMACCMPilot. https://smaccmpilot.org/.

[35] Google Inc. 2018. Android Things. https://developer.android.com/things/index.html.

[36] Michael S. Gordon, James R. Kozloski, Peter K. Malkin, and Clifford A. Pickover. 2015. Autonomous Drone Service System. US Patent Application no. US20160306355A1.

[37] Vivien Gueant. 2015. iPerf - The TCP, UDP and SCTP Network Bandwidth Measurement Tool. https://iperf.fr/.

[38] HobbyKing. 2017. Turnigy nano-tech 5000mah 3S 35 70C Lipo Pack w/XT-90. https://hobbyking.com/en_us/turnigy-battery-nano-tech-5000mah-3s-35-70c-lipo-pack-xt-90.html.

[39] Alexander Van't Hof, Hani Jamjoom, Jason Nieh, and Dan Williams. 2015. Flux: Multi-Surface Computing in Android. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys 2015)*. Bordeaux, France, 24:1–17.

[40] Jim Drew. 2014. JR Module in Taranis - Latency Testing. https://www.rcgroups.com/forums/showthread.php?2169724-JR-Module-in-Taranis-latency-testing.

[41] John Kos. 2015. TX/RX Latency Test Results. https://web.archive.org/web/20160901051340/http://rc.runryder.com/helicopter/t172571p1/.

[42] Eugene Kim, Jinkyu Lee, and Kang G. Shin. 2013. Real-time Prediction of Battery Power Requirements for Electric Vehicles. In *Proceedings of the 2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS 2013)*. Philadelphia, PA, 11–20.

[43] Anis Koubaa, Basit Qureshi, Mohamed-Foued Sriti, Yasir Javed, and Eduardo Tovar. 2017. A Service-Oriented Cloud-Based Management System for the Internet-of-Drones. In *Proceedings of the 2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC 2017)*. Coimbra, Portugal, 329–335.

[44] Oren Laadan and Jason Nieh. 2007. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007)*. Santa

Clara, CA, 323–336.

[45] Oren Laadan and Jason Nieh. 2010. Operating System Virtualization: Practice and Experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference (SYSTOR 2010)*. Haifa, Israel.

[46] Jintack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. 2017. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017)*. Shanghai, China, 201–217.

[47] Linux Kernel Organization Inc. 2016. Real-Time Linux Wiki. https://rt.wiki.kernel.org/index.php/Main_Page.

[48] Sara Mahmoud, Nader Mohamed, and Jameela Al-Jaroodi. 2015. Integrating UAVs into the Cloud Using the Concept of the Web of Things. *Journal of Robotics* 2015 (Jan. 2015), 10:1–10:10.

[49] Sara Yousef Mohamed Mahmoud and Nader Mohamed. 2015. Toward a Cloud Platform for UAV Resources and Services. In *Proceedings of the 4th IEEE Symposium on Network Cloud Computing and Applications (NCCA 2015)*. Munich, Germany, 23–30.

[50] Monsoon Solutions Inc. 2016. Monsoon Solutions Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/.

[51] OpenVZ. 2018. Checkpoint/Restore in Userspace. http://www.criu.org.

[52] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*. Boston, MA, 361–376.

[53] PassMark Software Inc. 2017. PassMark PerformanceTest - Android Apps on Google Play. https://play.google.com/store/apps/details?id=com.passmark.pt_mobile.

[54] Shaya Potter. 2010. *Virtualization Mechanisms for Mobility, Security and System Administration.* Ph.D. Dissertation. Columbia University.

[55] Shaya Potter and Jason Nieh. 2010. Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*. Boston, MA, 103–116.

[56] Shaya Potter and Jason Nieh. 2011. Improving Virtual Appliance Management through Virtual Layered File Systems. In *Proceedings of the 25th Large Installation System Administration Conference (LISA 2011)*. Boston, MA, 25–38.

[57] Qualcomm. 2016. Leading the World to 5G: Evolving Cellular Technologies for Safer Drone Operation. https://www.qualcomm.com/media/documents/files/leading-the-world-to-5g-evolving-cellular-technologies-for-safer-drone-operation.pdf.

[58] Qualcomm. 2017. LTE Unmanned Aircraft Systems Trial Report. https://www.qualcomm.com/documents/lte-unmanned-aircraft-systems-trial-report.

[59] Raspberry Pi Foundation. 2016. Camera Module V2 - Raspberry Pi. https://www.raspberrypi.org/products/camera-module-v2/.

[60] Raspberry Pi Foundation. 2017. Raspberry Pi 3 Model B. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[61] Raspberry Pi Foundation. 2018. Raspbian. https://www.raspberrypi.org/downloads/raspbian/.

[62] T-Motor. 2017. 6th Anniversary Limited Edition T-Motor Combo Pack. http://store-en.tmotor.com/goods.php?id=453.

[63] The Linux Foundation. 2017. Cyclictest. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest.

[64] Chien-Ming Tseng, Chi-Kin Chau, Khaled M. Elbassioni, and Majid Khonji. 2017. Flight Tour Planning with Recharging Optimization for Battery-operated Autonomous Drones. *CoRR* abs/1703.10049 (2017). arXiv:1703.10049

[65] United Parcel Service. 2017. UPS Tests Residential Delivery Via Drone Launched From atop Package Car. https://pressroom.ups.com/pressroom/ContentDetailsViewer.page?ConceptType=PressReleases&id=1487687844847-162.

[66] U.S. Department of Transportation. 2018. FAA Drone Registry Tops One Million. https://www.transportation.gov/briefing-room/faa-drone-registry-tops-one-million.

[67] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. 2017. FarmBeats: An IoT Platform for Data-Driven Agriculture. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017)*. Boston, MA, 515–529.

[68] Igor Vereninov. 2016. Issue with Navio2 with a Potential Flyaway Condition. https://community.emlid.com/t/issue-with-navio2-with-a-potential-flyaway-condition/2179.

[69] Amos Waterland. 2014. Stress Project Page. https://people.seas.harvard.edu/~apw/stress/.

[70] Justin Yapp, Remzi Seker, and Radu Babiceanu. 2016. UAV as a Service: Enabling On-Demand Access and On-the-Fly Re-Tasking of Multi-tenant UAVs Using Cloud Services. In *Proceedings of the 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC 2016)*. Sacramento, CA, 1–8.