Team-level Programming of Drone Sensor Networks

Luca Mottola[†*], Mattia Moretta[†], Kamin Whitehouse[‡], and Carlo Ghezzi[†]
[†]Politecnico di Milano (Italy), [*]SICS Swedish ICT, [‡]University of Virginia (USA)

Abstract

Autonomous drones are a powerful new breed of mobile sensing platform that can greatly extend the capabilities of traditional sensing systems. Unfortunately, it is still non-trivial to coordinate multiple drones to perform a task collaboratively. We present a novel programming model called *team-level programming* that can express collaborative sensing tasks without exposing the complexity of managing multiple drones, such as concurrent programming, parallel execution, scaling, and failure recovering. We create the VOLTRON programming system to explore the concept of team-level programming in active sensing applications. VOLTRON offers programming constructs to create the illusion of a simple sequential execution model while still maximizing opportunities to dynamically re-task the drones as needed. We implement VOLTRON by targeting a popular aerial drone platform, and evaluate the resulting system using a combination of real deployments, user studies, and emulation. Our results indicate that VOLTRON enables simpler code and produces marginal overhead in terms of CPU, memory, and network utilization. In addition, it greatly facilitates implementing correct and complete collaborative drone applications, compared to existing drone programming systems.

Categories and Subject Descriptors

D.3.2 [**Programming languages**]: Language classifications—*Concurrent, distributed, and parallel languages*; C.3 [**Special-purpose and application-based systems**]: [Real-time and embedded systems]

General Terms

Languages, performance

Keywords

Drones, programming, teams, sensing, coordination

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SenSys'14, November 3–5, 2014, Memphis, TN, USA.
Copyright 2014 ACM 978-1-4503-3143-2/14/11 ...$15.00
http://dx.doi.org/10.1145/2668332.2668353

1 Introduction

Autonomous drones are emerging as a powerful new breed of mobile sensing system: small embedded computers that move autonomously while carrying rich sensor payloads, such as cameras and microphones [13]. Aerial, ground, and aquatic drones are available off-the-shelf today and often come with a simple abstraction for navigation [1, 15, 22]. Many drones can be controlled by setting waypoints or by manually steering using a graphical interface through a tablet or a smartphone. As new designs emerge, drones continue to achieve higher speeds, carry larger payloads, and travel longer distances on batteries.

Drones can greatly extend the capabilities of traditional sensing systems while simultaneously reducing cost. They can monitor a farmer's crops [46], manage parking spaces [16], or monitor underwater telecommunication systems [15] more practically and/or more cheaply than stationary sensors. Compared to other kinds of mobile sensing, such as opportunistic sensing that piggybacks on the mobility of phones or vehicles, drones offer *directly control* over *where* to sample the environment: the application can explicitly instruct them on where to move. Thanks to this, they can even perform sensing tasks that had previously been beyond reach, such as collecting high-resolution imagery of civil infrastructures or inside forests where neither satellites nor vehicle-mounted cameras can reach [8, 33].

Problem. We aim to enable the coordination of multiple drones with a specific focus on *active sensing* applications, where the goals of the system evolve on-the-fly as new data is collected, and so the drones must be dynamically re-tasked. For example, active sensing may be used to follow moving environmental phenomena, such as a pollution cloud, or to sample more densely where specific conditions occur, such as where the pollution levels are highest [45]. Because the lifetime of most drone platforms is dictated by the execution speed, such tasks must be completed as quickly as possible.

Although deploying multiple drones may, in principle, improve the execution speed in many applications [28], coordinating multiple drones to complete some sensing tasks collaboratively is still non-trivial. There are currently two ways to achieve this: *i)* drone-level programming; and *ii)* swarm programming. *Drone-level programming* requires each drone to be given a sequence of actions to perform, including navigation, communication, and sensing. This approach can express almost any collaborative behavior, but requires programmers to individually address each drone, and

to explicitly deal with concurrency and parallelism, drone failures, navigational error, and the possibility of drones being added or removed. These issues greatly complicate the programming task.

On the other hand, *swarm programming* [11] explicitly forbids shared or global state: all drones execute a single set of basic rules and operate only on their own local state. This approach is easy to program and trivial to scale up to multiple drones, but can only express emergent swarm properties such as dispersion or flocking; it cannot achieve tasks that require explicit drone coordination, such as sampling a sensor at multiple locations simultaneously.

Contribution. We present a new drone programming model called *team-level programming*, whose conceptual novelty lies in creating a middle-ground between drone-level and swarm programming. In team-level programming, users express sophisticated collaborative sensing tasks without resorting to individual addressing and without being exposed to the complexity of concurrent programming, parallel execution, scaling, and failure recovery.

The basic approach is to allow the user to define the set of sensing tasks that must be completed, subject to constraints in space and time. Under the hood, the "team" run-time system automatically chooses the actions for each drone that will collaboratively accomplish the sensing tasks. Because the "team" does not choose individual drone actions until run-time, the system can easily scale to an arbitrary number of drones without requiring code modifications. It can also automatically adapt the execution plan as drones crash or deviate from their flight paths, perhaps due to obstacles or environmental effects, such as wind or water currents.

The main benefits of this approach are enabled by the fact that the user only specifies the sensing tasks without expressing the details of how individual drones execute the tasks; these details are chosen by the run-time system. However, this is also the source of the approach's main limitation: since the user cannot address individual drones, she cannot express actions that involve direct interactions between drones, such as those to pass a ball between drones [43].

The team-level abstraction is thus most suitable for collaboratively achieving a set of independent actions that could, in principle, be achieved even by a single drone, but for constraints on battery life, speed, and sensing range are more effectively achieved by deploying multiple drones. We argue that this is a useful layer of abstraction because the vast majority of mobile sensing applications where drones may be employed fall in this category. We return to this subject in Sec. 2 when we contrast our work with the state of the art, and in Sec. 3 where we describe a representative application.

To explore the concept of team-level programming, we create a programming system called VOLTRON, described in Sec. 4. In contrast to most distributed systems where communication is the parallism bottleneck, the bottleneck in collaborative drone applications is physical navigation. VOLTRON offers several programming constructs to offer the illusion of a simple sequential execution model while still maximizing opportunities to dynamically re-task drones to meet programmer-provided spatio/temporal constraints. The constructs include a notion of *abstract drone* used as a single entry-point to the functionality of the entire team, and variables enriched with *spatial semantics*. In addition, programmers define *time assertions* to state temporal constraints on sensed data that the run-time system must meet when scheduling the drone operation. As described in Sec. 5, the VOLTRON programming constructs can be translated to executable code in mainstream languages.

To concretely experiment with VOLTRON, we implement a system prototype running on the AR.Drone 2.0 quadcopter drones, as illustrated in Sec. 6. We choose the AR.Drone 2.0 merely because it is both cheap and easily accessible, also when it comes to spare parts, which facilitates the experimental activities. While the team-level programming model and its realization in VOLTRON remain independent of the specific type of drone—being it aerial, ground, or aquatic, for example—our current implementation could be extended to the latter platforms by swapping in the appropriate navigation engine. On the other hand, the navigational tools for aquatic and land-based drones are often not as market-ready as the aerial drone ones.

We report in Sec. 7 on the evaluation of VOLTRON based on three representative applications: aerial photogrammetry [33], building 3D maps of pollution concentration, and aerial surveillance. We perform a user study with 24 junior programmers to compare drone-level, swarm, and team-level programming. The results indicate that the latter is the only method that allows junior programmers to create both correct and complete collaborative drone programs. Their opinions, collected through a dedicated questionnaire, also point to the ease of use of VOLTRON compared to the alternatives. We also implement all three applications using drone-level, swarm, and team-level programming. We evaluate their performance using real-world experiments and an emulation testbed that can scale up to 100 drones. Our analysis indicates that VOLTRON produces marginal overhead in terms of CPU, memory, and network utilization, while transparently and dynamically scaling to different numbers of drones.

Finally, using our prototype we perform a real deployment of aerial photogrammetry [33] at a 32,000 m^2 archaeological site in Aquileia (Italy), as we describe in Sec. 8. The team of drones was able to produce maps of the site with distortion levels within 3.3% by dynamically adapting to the number of drones, changing sunlight levels and cloud movements, and varying wind forces from 0-8 knots. In terms of scaling, it automatically decreased total execution time by 3-3.5x as it ran with 1 drone to 7 drones, completing the task in a fraction of the time needed by current practices.

2 Background & Related Work

Mobile sensing is quickly expanding to a diverse set of applications and platforms. Popular examples are found in applications that leverage the mobility of phones [23] and vehicles [27], or in scenarios where sensors are hooked to entities that move autonomously, such as animals [29, 38]. A characteristic common to these examples is the fact that sensor movements are outside of the programmers' control. These instruct the devices to sample sensor data in an opportunistic manner, while mobility can only be passively monitored by the application. Several systems aim at simplifying programming in these scenarios [23, 25, 29]. When using autonomous drones, on the other hand, a device mobility becomes *part of the application logic*, and must be explicitly

encoded. This requires to address completely different challenges, such as how to specify the device movements and how to split the sensing tasks among multiple devices subject to spatio/temporal constraints.

The landscape of current drone platforms is a diverse one. Existing devices often contain quite powerful computing resources. Many inexpensive ($200 to 1K$) platforms today are built around variants of the PX4 autopilot boards [39], which applies to aerial, ground, and aquatic drones. These platforms typically feature ARM Cortex MCUs with FPU acceleration and megabytes of RAM, combined with high-speed networking via WiFi or cellular. Although a significant part of the computing power is consumed by the autopilot functionality, the remaining capacity can be used for application-level functionality via a POSIX-like RTOS.

The dominant practice in programming drones is to create a set of pre-defined commands combined into a single script and programmed onto each drone. For example, the ground drone iRobot Create [22] offers a low-level interface to control the drone movements with bindings available for several programming systems such as Player [21]. Based on this, software packages are built to offer basic functionality such as waypoint navigation [24, 36]. However, this approach quickly becomes unmanageable as the number of simultaneously-deployed drones increases. Programmers must manually decompose the goal into a set of single-drone parallel tasks. Moreover, to fully exploit parallelism, they must estimate the duration of each task and balance the load between drones, while taking temporal constraints into account. This analysis is complicated because the timing of a drone's actions depends on unpredictable factors, such as obstacles and crashes.

Real-time communication among drones can be used to collaboratively adapt and respond to these conditions. Inter-robot communication libraries do exist [28, 42], but moving from independent parallel operation to system-wide coordination is also difficult, because it requires parallel programming mechanisms such as data sharing, synchronization, and deadlock avoidance. As a result, most drone applications today use only a small number of drones at a time even when more drones would improve performance, or evaluate larger-scale cooperative strategies only in simulation [12, 40].

Differently, in robot *swarms*, the drones operate only on node-local state [11, 28]. In these systems, the programmers' commands are translated into a sequence of primitive instructions deployed onto all drones. Simple drone behaviors are shown to produce emergent swarm properties, such as dispersion or flocking, but cannot achieve tasks that require explicit coordination, such as sampling a sensor at different locations within time constraints. Several programming systems are applicable in this area. The Robot Operating System (ROS) [42] provides a Publish/Subscribe coordination layer for decentralized computations; Karma [17] lets programmers specify modes of operation for the swarm, such as "Monitor" or "Pollinate"; and Proto [6] lets programmers specify actions in space and time. Meld [4] provides similar concepts for modular robotics. In contrast to reasoning in terms of decentralized computations based on local states, we wish to provide a holistic perspective that allows to explicitly encode actions for the entire ensemble of drones.

Compared to macroprogramming sensor networks, our work considers a form of mobility that existing systems do not typically address [31]. This impacts both the programming model and the distributed execution. As for the former, being a drone's movements part of the application logic, they need to be explicitly encoded rather than passively observed, as it happens in many mobile sensor network applications [29]. In addition, the use of location information is not limited to reconstruct environmental phenomena [34], but are used to proactively instruct the system on where the sensing operation must take place.

On the other hand, during execution, communication among drones may happen in real-time rather than in a delay-tolerant manner [47]. Moreover, the run-time system needs to consider each drone individually instead of treating all sensor nodes equally because each drone requires individual, but coordinated path planning. Finally, dealing with parallelism and concurrency is markedly different. In mainstream sensor networks, it is network communication and input/output operations that need to occur in parallel with data processing. In our context, it is the drones' movements, combined with the inherent unpredictability of the navigation time, that makes the concurrency problem complex.

Finally, in the specific field of aerial drones, demonstrations exist that do show drones performing sophisticated collaborative tasks, but only for a handful of one-off applications created by skilled developers. For example, collaborating quadcopters are able to throw and catch balls [43] and carry large payloads [30]. In these settings, a centralized computer system communicates with the drones 100 times per second or more, and the goal is mainly to explore new approaches for mechanical motion control. The code to execute these tasks is indeed written by experts and custom-tailored to the specific application from the low-level drone control up to the necessary inter-drone coordination.

3 Example Application

In contrast to existing literature, in this work we provide a generic coordination substrate usable as-is for a large set of multi-drone applications. These generally show some distinctive traits: *i)* they require the drones to sample the environment subject to variable spatio/temporal constraints; *ii)* the workload, which in principle only one device may carry out, can be split among multiple drones; *iii)* as a result, the total execution time can be reduced by deploying additional drones; and *iv)* direct interactions between drones that would require individual addressing are not needed. In scenarios with these characteristics, our work spares much of the effort required by building upon the low-level drone APIs, and automatically and transparently manages parallelism as the number of drones is scaled up or down.

An illustrative example is that of aerial mapping of archaeological sites using photogrammetry techniques [33]. Aerial drones are increasingly employed in these scenarios because physically walking on or near the site can cause irreparable damage. In this field, we are collaborating with archaeologists at the university of Trieste (Italy) to survey a site called *Domus dei Putti Danzanti* [19]. The 32,000 m^2 area, partly shown in Fig. 1, hosts the ruins of an ancient Roman house dating to the fourth century BC. The excavations are bringing back to life the layout of the house and its

Figure 1. *Domus dei Putti Danzanti* archaeological site in Aquileia (Italy)—NE corner.

surroundings. Archaeologists rely on this information to reconstruct the social interactions of the era; for example, how the owners—conjectured to be wealthy businessmen—used to treat their servants. This can be determined by conjecturing about the usage of the different spaces in a house based on shape and size.

We use VOLTRON to support this study and present the results as part of our deployment experience. Specifically, to provide the geometric layout of the *Domus dei Putti Danzanti*, we use a team of aerial drones to derive *orthophotos* of the site [35]. An orthophoto is an aerial photo that is geometrically-corrected so that distances between pixels are proportional to true distances, such that the photo can be used as a map. A series of small aerial pictures is stitched together to derive a single orthophoto by correcting for distortion effects [35]. The resulting accuracy is mainly determined by the amount of overlap between the single pictures.

Currently, drone operators run the orthophoto software once a drone has completed a full scan of the area [33]. If the individual pictures do not have sufficient overlap, however, the resulting orthophoto will show excessive aberrations. The drone is then sent out again to gather pictures at a finer granularity. The more pictures a drone needs to acquire, the more time it takes to complete the full scan. At the same time, the scan must be completed within a limited time because the scene may change; for example, due to varying lighting conditions caused by shadows or moving clouds. If the time lag between any two pictures is excessive, the information will be inconsistent and will affect the accuracy.

Archaeologists thus need a simple way to program a team of drones to scan an area, to continuously evaluate the aberration levels of the orthophoto based on the obtained overlapping, and to accordingly adapt the drone operation at runtime. The team of drones should automatically deal with challenges such as wind and collisions among them. Also, if the archaeologists are unable to produce an adequate orthophoto within the allowed time, they should be able to add more drones and the system will automatically re-balance the scanning tasks to achieve a higher quality orthophoto.

We choose this application as a running example because the requirements for active sensing, scalability, and ease of programming are representative of the applications we target. Indeed, even though a single drone may carry out the entire scanning of the site subject to the above spatio-temporal constraints, multiple coordinating drones may split the work to reduce the execution time and/or obtain more accurate orthophotos.

4 Programming Constructs

We describe the design of VOLTRON: a set of programming constructs to explore the notion of team-level drone programming. In a nutshell, we introduce: *i)* a notion of *abstract drone* that allows the code to remain unchanged independent of the number of deployed drones; *ii)* spatial semantics for variables and a **`foreachlocation`** loop to enable parallel and independent actions to be requested for a given set of locations; *iii)* a custom system of *futures* and *promises* [20] to facilitate dynamic (re)scheduling of drone operation; and *iv) time assertions* to state constraints on sensed data.

Unlike sensor network macroprogramming, the notions *i)* and *ii)* above are required to explicitly encode the drone movements as part of the application logic. Moreover, *iii)* is required because, in contrast to other kinds of distributed computing, the drone movements represent the main factor potentially hampering parallelism. Finally, *iv)* allows programmers to *declaratively* control the time evolution of drone operations, which is typically not required in other kinds of mobile sensing.

We use the application from Sec. 3 for illustration, but the abstractions are more generally applicable. In Sec. 7, we describe several other applications that we implement with VOLTRON. Our design assumes that: *i)* drones have access to their own location; for example, from an on-board GPS module; *ii)* drones have access to a common time base; for example, through a time synchronization protocol; and *iii)* drones can wirelessly communicate with each other and with any ground-station that might exist. The vast majority of existing drone platforms can or do meet these assumptions.

4.1 Abstract Drone

We define an abstract device with custom APIs that programmers can use to task the drones without individual addressing. This device is called **`Drones`**. As the abstract drone is the only entry point to the system's functionality, an application's code can remain unaltered no matter how many real devices are deployed.

Fig. 2 shows the API of the abstract drone. It mainly offers a **`do`** command that accepts as *inputs*: *i)* the name of an action; for example, **`"pic"`** to take a picture; *ii)* a set of physical locations where to perform the action; for example, the points of a (possibly time-varying) spatial grid; *iii)* optional parameters to customize the execution; for example, what sensor the drone needs to use to carry out action **`"pic"`**; and *iv)* a handle to retrieve a reference to the executing action. Based on these inputs, the abstract drone transparently coordinates the real devices so that the given action is eventually performed at every input location. The abstract drone can achieve this as it sees fit; for example, if there are more locations than drones, it can freely schedule the drones to rotate through the different locations.

The *output* of the **`do`** command is a multiset of values of the same data type, each representing the result of performing the given action at a specific location. For example, the output of performing action **`"pic"`** over a 3 x 3 x 3 grid is eventually going to be a multiset of 27 values of type **`Image`**, each associated with a different location among the 27 in the grid. The multi-set is progressively constructed as the drones perform the given action at the desired locations.

Operation	Inputs	Description
do	**action** (singleton) **locations** (set $\neq \emptyset$) **parameters** (set) **handle** (singleton)	Perform an **action** at specific **locations**, customized by **parameters**, linked to **handle**.
stop	**handle** (singleton)	Stop the running action linked to **handle**.
set	**key** (singleton) **value** (singleton)	Set a $\langle key, value \rangle$ pair in the registry.
get	**key** (singleton)	Read the value of *key* from the registry.

Figure 2. Abstract drone API.

The abstract drone additionally offers a **stop** operation to cancel previous **do** commands, and **get/set** operations to read or write from a registry of $\langle key, value \rangle$ pairs that stores application parameters and system information. The API is intentionally kept generic to be platform-independent. We acknowledge that this design decision makes it harder to manage heterogeneous systems of drones with different capabilities or roles. Different teams can be separately controlled with independent abstract drones, but parallelism and coordination between teams would be left to the programmer. This should still be easier than programming each drone individually, and it remains an opportunity for future work.

4.2 Reasoning with Space

The role taken by space information in VOLTRON is different than in other sensing scenarios. Indeed, location information are not opportunistically acquired and later used to process sensor data, but directly determine where the sensors are going to be sampled. Because of this, we give variables a spatial semantics both to specify where drones are to move to sample the environment, and to encode the mapping between the results of actions and the associated locations.

Spatial variables. Fig. 3 reports a fragment of VOLTRON code implementing the core functionality in the archaeological site application. The VOLTRON constructs are added as extensions to an existing "host" language, Java in this case. We discuss this choice in Sec. 7. The drones inspect the site at increasingly higher granularity based on an estimate of the aberration in the resulting orthophoto. This is computed at run-time using available orthophoto libraries [32][1].

Line 1 in Fig. 3 declares a spatial variable **tiles** whose data type is **Image**. The **spatial** qualifier indicates that the variable can store *one or more* **Image** values each associated to a different location. These values are going to be the individual aerial pictures that we eventually assemble in a single orthophoto. Generally, spatial information act as an additional facet in variable declarations, orthogonal to the data type. This allows VOLTRON to retain the type system of the host language.

Lines 2 to 15 implement the active sensing loop. Line 4 assigns to **tiles** the spatially-associated values resulting from the execution of the abstract drone over a set of locations arranged as a **grid** of a given granularity. The action is to take a picture at every input location. The **grid** is input to the abstract drone to specify where to carry out the action, and is separately specified as explained next. The **@** operator in variable assignments thus only permits the write of values

[1] These kind of estimations do *not* require to compose the actual orthophoto, and so are generally sufficiently lightweight to run also on embedded platforms.

```
1  spatial Image tiles;
2  do {
3   tilesOK=true;
4   tiles@grid=Drones.do("pic",Drones.cam,handleCam);
5   foreachlocation (Location loc in tiles){
6    float aberr=abberationEstimate(tiles@loc);
7    if (aberr > MAX_ABR && !sceneChanged) {
8     Drones.stop(handleCam);
9     int newStep=tune(Drones.get("gridStep"),aberr);
10    Drones.set("gridStep",newStep);
11    tilesOK=false;
12    break;
13   }
14  }
15 } while (!tilesOK);
```

Figure 3. Adaptive sampling of aerial pictures for orthophoto processing.

```
1 boolean grid(Location location) {
2  if (location.x % Drones.get("gridStep")==0
3      && location.y % Drones.get("gridStep")==0)
4   return true;
5  else return false;
6 }
```

Figure 4. Specification of a grid geometry. Location is a built-in data type, customized depending on the specific localization system.

associated to the given specific location(s).

The location(s) used as parameter for the **@** operator can be single locations or specified in a functional manner within a *finite* coordinate system. Fig. 4 shows an example of the latter for the **grid** geometry. When executing an assignment with a space geometry, the system enumerates all possible locations—based on a platform-specific unit metric—and checks the return value of the space geometry for each of them. If this is **true**, the location is given as input to the abstract drone[2].

Concurrency and loops. Calls to the abstract drone may simply be blocking. However, drone operations take time. Moreover, as our application illustrates, the data the drones harvest rarely needs to be processed at once. For example, to compute the expected aberration of the resulting orthophoto, we can process the pictures as they come. Based on this, programmers can dynamically adapt the grid granularity.

To enable higher concurrency between drone operations and data processing, one may straightforwardly employ parallel threads with proper synchronization. However, parallel programming is challenging in general and, as we demonstrate in Sec. 7, it likely leads to programming errors in our context. Based on this, to enable higher concurrency and yet retain the sequential semantics programmers are most familiar with, we adopt a custom system of *futures* and *promises* [7, 20] for spatial variables.

Whenever a spatial variable is assigned to the result of a **do()** operation, we create a future for every location eventually holding the action's result. The future acts as a "placeholder" waiting for the drones to provide the concrete value, that is, to "complete" the promise. Intuitively, the **do()** call on line 4 in Fig. 3 creates a "grid of futures", one for every

[2] The format of location information is, in general, platform specific, and here specified as $\langle x, y \rangle$ coordinates merely for simplicity. The system also allows one to configure a tolerance margin whereby a location is considered reached anyways.

location that function **grid** in Fig. 4 considers. This allows the execution to continue until the value becomes necessary. Only then, the execution blocks waiting for the promise.

We complement this design with a custom **foreachlocation** loop that allows the programmer to iterate over the results of the abstract drone as they come, using the associated locations as an index, as in line 5 of Fig. 3. A **foreachlocation** loop can process the data *independently* of the order these data is stored in memory. Specifically, whenever concrete values are available in a spatial variable, the loop iterates over those locations first. The loop blocks when every concrete value is already processed, but more futures are set for the spatial variable. It then unblocks as some of these futures are replaced by the corresponding promises, until no more futures are set.

In Fig. 3, the **foreachlocation** in line 5 likely blocks initially until the first pictures arrive. As this happens, the loop unblocks. Next, we pass the obtained pictures to an orthophoto library to compute the estimate of the final aberration. The execution proceeds differently depending on whether the estimate exceeds a predefined threshold and the scene has hitherto not changed (line 7). The latter requirement is a function of temporal aspects; we illustrate in Sec. 4.3 how we check this situation. If either of the conditions is not met, the loop rewinds until it blocks again waiting for the drones to provide more pictures.

Differently, we stop the abstract drone (line 8), adapt the grid granularity based on current step and expected aberration (line 9 and 10) and set a flag (line 11) that causes the abstract drone to reschedule operations with the new grid. We intentionally do *not* reinitialize the **tiles** variable. Some locations in the new grid might be the same as in the earlier one. The abstract drone may not need to re-do the work for those locations. By not re-initializing the variable, the abstract drone finds some already-completed promises in **tiles** that it can simply skip.

Uncompleted promises. Drones are failure-prone [13]. Moreover, even a fully-functional system may trivially fail because of environment constraints; for example, obstacles preventing the drones to reach the desired locations. These situations entail that the drones may be unable to complete some promises, eventually bringing the execution to a halt.

To cope with these situations, we provide an **await** construct to unblock waiting for a promise, as in:

```
foreachlocation (Location loc
                 in tiles await 10MIN){//...}
```

where the execution blocks at the loop for at most ten minutes before continuing anyways. Should the timeout fire before the drones complete any of the promises in **tiles**, the loop unblocks anyways. Programmers can determine whether the timeout fired for a future, or simply a future at a given location does not exist, using the **isfuture** keyword, as in:

```
if (isfuture tiles@arbitraryLoc)
  System.out.println("The timeout fired.");
else if (tiles@arbitraryLoc == null)
  System.out.println("No future at " + arbitraryLoc);
```

Note that programmer can attach the **await** constructs to any read operation of spatial variables.

```
1 time_assert maxLag(spatial Image tiles) {
2  forall locA:Location in tiles,
3  forall locB:Location in tiles |
4    Math.abs(T?tiles@locA - T?tiles@locB)>10MIN
5  on violate { sceneChanged=true; }
6 }
```

Figure 5. Time assertion to stop adapting the grid granularity if the scene should be considered as changed.

4.3 Reasoning with Time

Many drone applications are time-sensitive, especially when the data the drones harvest need to be obtained within given temporal bounds, as in our example application. This contrasts other kinds of mobile sensing where the processing is often delay-tolerant. VOLTRON deals with such requirements by giving programmers ways to express time assertions. We opt for a declarative approach in that the temporal requirements at stake are most naturally expressed this way, and yet it is practically possible to transform such declarative specifications in the necessary procedural code.

In our example application, should the time interval between any two pictures cross a threshold, programmers must consider the scene as changed and stop adapting the grid granularity. It is now more important to complete the inspection, because the changes in the scene will anyways mainly affect the accuracy of the resulting orthophoto. The specification in Fig. 5 implements the relevant time constraint. Programmers access temporal information via a **T**? operator to state the actual constraint in line 4. The fragment of code within the **on violate** (line 5) clause is executed as soon as a violation to the time assertion is detected.

VOLTRON time assertions are, in essence, predicates defined over the timestamps of at least one quantified spatial variable. Thus, their general form is:

```
(forall | exists) location_name:Location
                  in spatial_variable_name
(,(forall | exists) location_name:Location
                    in spatial_variable_name)+ |
  time_predicate
  on violate {\\...}
```

which arguably covers most of the expected temporal requirements in the applications we target.

The assertions may refer to arbitrary spatial variables identified *based on their name*. The parameter **tiles** in line 1 of Fig. 5 refers to a variable with this name visible from where the time assertion is defined. Our translator converts the time assertions in a side effect-free functional equivalent, scheduled in parallel with the main program. In addition, it reserves a special handling to time assertions whenever they can be considered to further optimize the drones' movements, as we describe in Sec. 6.

Programmers can use the **T**? operator also outside of time assertions. For example, in the application of Sec. 3, we use it to compute the average time elapsed between consecutive pictures, useful to obtain a run-time estimate of the total execution time.

5 Compile-time Code Translation

We implement two source-to-source translators for VOLTRON programs, using Java or C++ as host language. Both translators convert VOLTRON code into programs written solely in the host language. These are handed over to

standard tool-chains for compilation.

Both translators are based on ANTLR and proceed the same way. First, they parse the input program to construct an abstract syntax tree (AST). Next, some nodes in the AST are converted using predefined translation rules. These procedures happen off-line and automatically. We define translation rules for every constructs we hitherto introduced. These are, in most cases, straightforward. Due to the impact they have on concurrency semantics, however, we briefly illustrate the translation for spatial variables and `foreachlocation` loops.

Spatial variables. We convert every spatial variable into a collection of structured data types. One instance of the structure includes a field of the original data type; for example, of type `Image` for the `tiles` variable in Fig. 3. It additionally includes a `Location` field representing the associated location, a flag indicating whether the variable holds a future, and an identifier pointing to the corresponding promise.

Every instance of these structured data types is inserted in a shared data repository that mediates read and write operations to spatial variables. Whenever the original program requires to read a spatial variable, the data repository looks up the specific value based on variable name and location. If the value is still a future, the read operation on the data repository blocks.

We translate every call to `do()` on the abstract drone to a specific sequence of operations. In case of a space geometry, the locations of interest are first collected by repeatedly querying its functional specification, such as `grid` in Fig. 4. Next, the locations are input to the abstract drone with the indication of the actions to take. Whenever the abstract drone completes a promise, we write the value to the shared data repository and every blocked process is scheduled again. This allows the original program to unblock on a previous read operation. An exception to this processing occurs when a future can be directly transferred to another spatial variable, as it happens in an assignment between spatial variables. In this case, we can copy the future to the assigned variable and continue.

Loops with `foreachlocation`. To provide the semantics described in Sec. 4.2, the order of visit of data referenced in the loop condition matters. In the example of Fig. 3, depending on the order the `foreachlocation` loop inspects the locations for variable `tiles` in the data repository, a read operation may block immediately on a future although concrete values exist somewhere else in the same collection.

To address this issue, the read operations of spatial variables in a `foreachlocation` loop are translated into a form of "conditionally-blocking" read through a circular buffer. Rather than blocking on the first future encountered, a read operation skips every future in the collection until it finds a concrete value yet to process. Conversely, the operation blocks if every concrete value is already processed and more futures are set in the same collection. At this point, the first promise that completes on this collection unblocks the process, and processing resumes from the value just written[3].

Figure 6. Custom AR.Drone with auxiliary embedded board.

6 Run-time System

The run-time performance of teams of drones is mostly determined by their execution time, that is, how much time it takes to complete the assigned tasks. Due to energy issues, the quicker a drone can complete a task, the fewer battery changes are needed, and thus the system becomes more practical.

Rationale. In VOLTRON, the implementation of the abstract drone encapsulates the mechanisms to drive the real devices. Crucially, we must determine the drones' movements *at run-time*. Off-line approaches whereby the sensing locations are divided beforehand among the available drones are unlikely to be sufficiently general and/or to work efficiently, as the a priori knowledge of the sensing locations may be limited. For example, it is generally impossible to pre-compute all locations returned by space geometries, as some information may only be available at run-time.

Perhaps the only general solution in this respect might be to split the field in non-overlapping sub-areas, and to assign all sensing tasks in a sub-area to a given drone. However, in general one cannot predict how such subsets of tasks would evolve based on sensed data. For example, sensor readings in one sub-area may cause the assigned drone to perform much more work than all others, leading to a non-optimal usage of resources.

Differently, using our prototypes, we achieve a form of inter-drone coordination that *continuously* balances the work among the available drones, adapting to the changes in the program state and to the dynamics of the environment. We do so in a centralized or state-replicated manner, as we describe next, with no changes required to the VOLTRON program in case it is to run using either execution model.

Target platform. We implement two prototypes in Java and C++, targeting customized AR.Drone 2.0 quadcopters [37], shown in Fig. 6. The AR.Drone is a cheap commercially-available quadcopter, equipped with an ARM Cortex A8 CPU, 128 Mb of RAM, a WiFi interface, two cameras, and a range of sensors for navigation. It runs an embedded version of Linux and comes with its own libraries for remote control. The team-level programming model and the language constructs of VOLTRON are independent of the specific type of drone and may be applied to other kinds of drones, such as ground or aquatic ones, by replacing the navigation engine. We choose the aerial drones because, among existing drone platforms, they arguably are the most-market ready, and are instrumental to the deployment effort we describe in Sec. 3.

We customize the AR.Drone by shaving off unnecessary

[3] Note that this implementation does not protect from concurrent modifications of other processes. As in standard Java, we expect programmers to take care of these situations explicitly.

payload, by adding another vertical camera, by replacing the battery with a higher-density one, and by interfacing an additional embedded board, shown in Fig. 6 mounted atop the drone. The board features an MSP430 MCU and an 802.15.4-compliant 2.4 Ghz CC2420 low-power radio. We use this board for integrating a LocoSys LS20031 GPS receiver (not shown) and for communication among drones.

6.1 Centralized Execution

A ground-station connected to the drones via WiFi centrally executes the VOLTRON program and stores the corresponding data repository. Hence, the drones store no application state, and every read/write operation happens at a single copy of the data. The ground-station orchestrates the drone's movements using platform-specific libraries for remote control [2]. The drones report their GPS location to the ground-station. Based on this and the inputs to the abstract drone, the ground-station decides how to move the drones.

Centralized executions are likely to be very efficient in terms of execution time, in that the ground-station determines the drones' movements based on global knowledge. On the other hand, the ground-station requires constant WiFi connectivity with the drones, which prevents them to operate outside of the ground-station WiFi range. Moreover, the ground-station is a single point of failure.

Despite these limitations, the centralized approach is useful in practice, as we discuss in Sec. 8, which illustrates our real-world experiences. Indeed, if needed, extending the ground-station WiFi range is practically quite simple: a cheap amplified directional antenna is sufficient to reach an AR.Drone more than 1 km away. Most importantly, being the fastest to execute, centralized executions require the least number of battery changes, as we show in Sec. 7, rendering the whole system much more practical.

Drone orchestration First, we must plan the drone movements w.r.t. each other. We must do so whenever a new `do()` call occurs, while also considering currently executing `do()` calls that may have not completed. As this occurs at run-time, we need a mechanism that quickly returns a solution, although sub-optimal.

We map the problem to a multiple traveling salesman problem (MTSP) [9], which generalizes the TSP in case of a finite number of salesmen (drones). This requires identifying a heuristic to split the input locations among the drones, then considering each subset of locations as input to a smaller TSP whose objective is the minimization of the traveled distance. For aerial drones, this is largely proportional to the execution time.

To split the target locations, one may apply some form of geographic balancing. For example, we may assign every drone to a subset of nearby locations. However, such a solution works efficiently only as long as the different subsets of locations can be covered in roughly equal time. This does not hold in general, because the environment may affect given portions of space more than others; for example, when an area is more exposed to the wind. If a drone is assigned such an "unlucky" portion of space, it may take longer to operate, while all other drones may have completed earlier and remain idle, again resulting in a non-optimal resource usage. This makes this kind of solution not sufficiently general. Therefore, we apply a different heuristic proven to work reasonably in practice [9], and randomly split the input locations in as many subsets as the number of drones. In the absence of further information, a random split is probably the only choice that evenly distributes the "unlucky" locations.

We solve the single TSP problems using the nearest neighbor greedy algorithm [9]. Next, we approximately evaluate each drone's traveling time based on distances between locations, the drone maximum speed, and the expected time for carrying out the action at every location. Provided a drone completes a promise at every location, we check every relevant time assertion. In case of violations, we discard this solution to the MTSP problem, and the process starts over with a different random division of the input locations. When a maximum number of iterations is reached, the programmer is notified that the system cannot find a navigation plan compliant with the time assertion. The programmer can then decide to interrupt the execution or to continue anyways.

Navigation. Once the navigation plan is determined for each drone, their concrete movements are driven by the Paparazzi [36] autopilot, running on the ground-station alongside the VOLTRON program. Potential collisions between drones are recognized at the ground-station based on the drones' positions. The ground-station prevents the collision by moving every drone only when at a safe distance to any other drone, or by making a drone hover until the surroundings are clear. We need to perform these checks at run-time, because merely identifying trajectory intersections does not suffice: the drones may unpredictably drift off the expected trajectories because of environmental phenomena such as wind gusts.

6.2 State-replicated Execution

To remove the single point of failure, every drone independently executes a separate copy of the program. To overcome the limitations of 1-hop WiFi connectivity, every drone directly communicates with others via the CC2420 radio, possibly across multiple hops, extending the overall range of operation. Compared to centralized executions, these advantages come at the price of sub-optimal decisions about the drones' movements, as each of them autonomously decides where to move.

State consistency. In this setting, the data repository is also replicated across the drones, and an obvious consistency problem arises: different drones may complete different promises at different times, so the program execution may differ across drones.

We employ a *virtual synchrony* replication model [10] to ensure that every copy of the program proceeds between the same sequence of states, and thus eventually produces the same output. Virtual synchrony is used for building replicated systems by totally ordering the inputs to different replicas. If the replicas behave deterministically, this ensures consistency of their outputs. Moreover, virtual synchrony can also deal with node failures, allowing the system to safely resume from failures occurring to some of the replicas.

Virtual synchrony ensures state consistency in our setting based on the observation that the *only* inputs to VOLTRON programs are the promises. Therefore, ensuring totally ordered updates to the shared data repository suffices to provide program consistency. When executing in this manner,

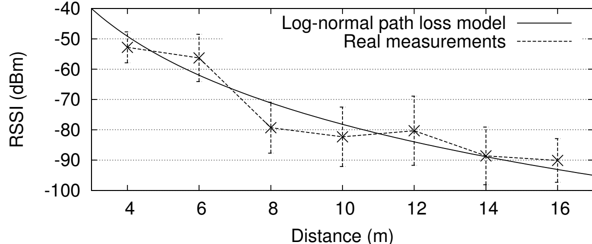

Figure 7. In the archaeological site, the CC2420 RSSI tends to follow the log-normal path loss model when drones are in flight.

calls to the data repository are negotiated by a virtual synchrony component. This component tags every call with a totally ordered sequence number. The operations are then executed according to their sequence number.

Supporting virtual synchrony in a highly mobile network is a challenge per se, and even more so in multi-hop networks. We tackle these issues by using Virtus [18]: a virtual synchrony layer recently developed for 802.15.4-compatible 2.4 Ghz radios. Virtus efficiently works across multiple hops while not relying on any topology information; hence, it is apt to operate in such highly mobile environment.

Coordinated navigation. Running the MTSP solver on the drones might not scale. We thus employ a simplified version of an existing multi-robot exploration algorithm [44]. In a nutshell, the algorithm drives the drones by creating a virtual potential field where the target locations are minima. Every drone autonomously moves towards the closest minima—identified by its GPS coordinates—by descending the corresponding gradient. It might then happen that two or more drones decide to reach the same minima. When the target location is reached, the minima is cleared. Updates to the virtual potential field are also propagated using Virtus, although total ordering for these is not mandatory: stale information may degrade performance; for example, whenever a drone moves towards an already cleared minima, but they do not affect overall consistency.

To prevent collisions, every drone creates virtual repulsion forces communicated to other drones using 1-hop beaconing from the CC2420 radio, so that far away drones are a priori excluded from processing. We deem the strength of the repulsion force proportional to the received signal strength indicator (RSSI) returned by the CC2420 radio, Because the drones stay away from ground, in the absence of obstacles and interference, the RSSI tends to follow known theoretical models. Fig. 7 exemplifies this based on micro-benchmarks in our archaeological site, calibrating the log-normal path loss model using linear regression.

The implementation of the navigation algorithm runs on the AR.Drone main board next to the VOLTRON program, interfacing directly with the AR.Drone firmware to issue the necessary flight commands and with the auxiliary board for communication with other drones. Our implementation is thus somewhat tailored to our custom hardware. However, many commercially-available drones are based on ARM platforms, as we mentioned in Sec. 2, while integrating the auxiliary board on a good fraction of them should be straightforward: a serial connection is all that is needed.

7 Evaluation

We evaluate VOLTRON in four different ways. In Sec. 7.2, we measure VOLTRON's run-time performance in real-world experiments to quantify its scalability in centralized and state-replicated executions. Sec. 7.3 reports the results of a user study involving 24 junior programmers, which demonstrates that VOLTRON eases the implementation of active sensing applications using multiple drones. Sec. 7.4 quantitatively corroborates this claim by evaluating the implementations based on established software metrics. In Sec. 7.5, we evaluate the cost of using VOLTRON in terms of system overhead in a large-scale emulated environment, and demonstrate that it does not appreciably impact the overall scalability.

7.1 Methodology

Applications. We evaluate VOLTRON using three representative applications. The first is the archaeological site application described in Sec. 3, which we call ARCH.

The second application, called PM_{10}, deals with building 3D maps of pollution concentration in the atmosphere [45]. The execution starts with a call to the abstract drone to sample this quantity along a predefined 3D grid. As this happens, we build a spatial profile of pollution concentration and progressively compute gradients towards areas of higher concentration. A second call to the abstract drone dispatches some of the drones to sample the pollution concentration along such gradients, enriching the spatial profile. The run-time system takes care of transparently allocating the available drones between the two concurrently-executing calls. Moreover, for the obtained spatial profile to be consistent, we must gather any two consecutive samples within a given time bound, which we express as a time assertion. In case of violations, the system adapts the sensor parameters to speed up the execution.

The third application, named PURSUE, is representative of surveillance applications [46]. The drones are to find and follow moving objects by providing an aerial picture of them whenever they first enter the camera field. The objects may appear or disappear unpredictably. Where to find an object is determined by sensor readings. The program includes two distinct operating modes. When in "patrolling" mode, the drones regularly inspect a predefined portion of space. When an object is found, some of the drones switch to "pursuing" mode. In our setup, the objects may move faster than the drones. This entails that not any single drone can constantly follow an object, but the abstract drone needs to dynamically switch between the real drones to ensure an object is constantly tracked. Time assertions are defined to ensure that drones meet a temporal upper bound between the detection of a moving object and when its picture is taken. In case of violations, every tracked object with at least one acquired picture is released from tracking to re-gain resources, lowering the acquisition latency for the next object.

Baselines. We compare VOLTRON with three baselines: *i)* drone-level programming, *ii)* swarm programming, and *iii)* a library-based version of VOLTRON. To perform the comparisons, we implement functionally-equivalent versions of our applications using the three systems described below.

We use the AR.Drone SDK as a representative of drone-level programming. Programs written with the AR.Drone

SDK must be executed on the ground-station and remotely control the drones. We implement the planning of the drones' movements using the same centralized planning algorithm described in Sec. 6.1.

We use ROS as a representative of swarm programming. ROS is established software and was recently ported to the AR.Drone [5]. The specific AR.Drone port does not support running all code on the drone itself; nonetheless, we encode the robot actions based only on local state in accordance with the principles of swarm programming. We implement the planning of the drones' movements using the same distributed planning algorithm described in Sec. 6.2.

Finally, we create a new implementation of VOLTRON that uses standard libraries instead of extensions to a host language. We call this implementation VOLTRONLIB. In library form, the abstract drone takes as additional input a reference to a listener object used to (asynchronously) process the data as it arrives. Spatial data are stored in standard collection objects. Programmers use concurrent threads to parallelize data processing and drone operation, with standard producer-consumer synchronization. Time assertions are specified in a procedural manner and manually scheduled concurrently with other threads.

In all three cases, when assessing the programming effort, we exclude the implementation of the navigation functionality from the analysis even though not all libraries fully provide it. This creates a worst-case setting for VOLTRON when it comes to comparing the different programming efforts.

7.2 Real-world Experiments

We conduct experiments for about one week at the archaeological site to evaluate how the scanning speed changes as the number of drones is increased. We compare both centralized and state-replicated executions.

Setting and metrics. We run ARCH with one to seven drones. We measure the application *execution time*, that is, the time it takes for a given number of drones to gather enough pictures within the given time constraint so that the resulting orthophoto is satisfactory.

Whenever a given number of drones cannot complete the inspection before the battery dies, the drones come back to the starting point, we replace the batteries, and we let them continue from where they left. In computing the execution time, we do not include the time for such battery replacement. Nonetheless, we also count the number of times this needs to happen as the number of *application rounds*. The drones move at a maximum speed of roughly 5 m/s. We repeat each experiment 5 times, and plot averages and standard deviations.

Results. Fig. 8 indicates that execution times are significantly reduced as the number of drones increases. In fact, only by deploying all seven drones in centralized executions can the 10-minute constraint in Fig. 5 always be satisfied, even if by a narrow margin. The speed increases diminish as more drones are added, as other factors come into play: *i)* the time the drones "waste" hovering to avoid collisions with other drones; and *ii)* the time necessary for inter-drone coordination, through either the ground-station or the virtual synchrony layer.

The diminishing returns could be ameliorated, at least

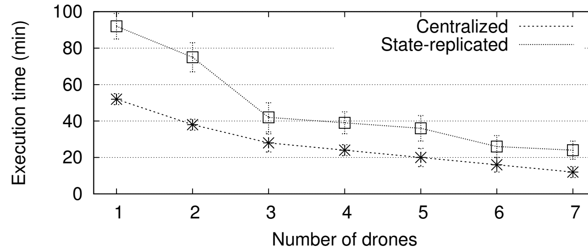

Figure 8. Using multiple drones provides significant improvements in execution time.

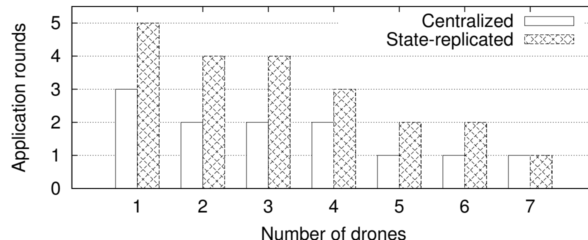

Figure 9. Five drones are sufficient not to replace batteries in centralized executions, seven drones are needed in distributed executions.

in centralized executions, by employing less conservative collision-avoidance mechanisms or smarter heuristics to split the input locations among the drones [9]. However, the processing times in the latter may render them impractical at run-time. We expect the execution times for the ROS and AR.Drone SDK implementations to be comparable to VOLTRON since they employ the same heuristic MTSP solution. We return in Sec. 8 to the the variability in execution time shown by the standard deviation in Fig. 8.

The same chart shows that, in general, state-replicated executions take more than twice the time of centralized executions. This is largely due to the sub-optimal choices of the exploration algorithm, and corresponds to the overhead for avoiding a single point of failure and extending the operating range. The decisions taken by the individual drones during state-replicated execution are similar to those during centralized ones, as long as drones are far apart. When two drones with overlapping paths come close together, however, the (virtual) repulsion forces described in Sec. 6.2 take longer to resolve the path conflict.

We also observe a constant penalty for state-replicated executions vs. centralized ones, visible even with only one drone. This appears because of two reasons: *i)* the exploration algorithm we use in the former works in discrete rounds: a drone moves for some time, stops to re-evaluate the state of the virtual potential field, and then resumes movement; hence, drones move slower than in centralized executions, where instead a drone heads directly to the destination; and *ii)* the virtual synchrony layer may take some time to recover lost packets [18]: this adds another source of latency, in that the exploration algorithm cannot resume movement until the state of the virtual potential field is updated. In Fig. 8, this penalty is somehow dominating until three drones are deployed, then part of it is amortized by the additional drones, but its influence still remains.

Fig. 9 depicts the number of execution rounds in our ex-

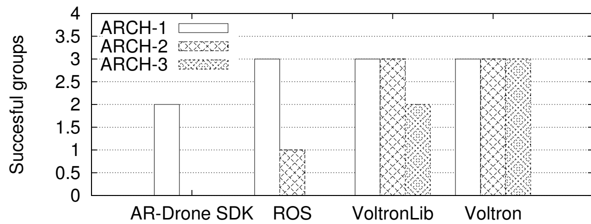

Figure 10. Only using VOLTRON all groups can successfully complete all programming exercises.

periments. These show minimal variability across different runs—not visible in the charts—and are almost exclusively a function of the number of drones. These figures give an intuition about the actual wall-clock time to execute the application, that is, including the time to replace the batteries. As an example, with only one drone, batteries need replacement three times in centralized executions, for a total of about 1.5 hours to complete a single application run. In centralized executions, five drones are sufficient to avoid replacing batteries altogether, whereas all seven drones are necessary in state-replicated executions.

7.3 User Study

We recruit 24 junior programmers and carry out a user study to compare the ease of use of several methods for programming teams of drones.

Settings. The programmers we recruit are M.Sc. students about to graduate at Politecnico di Milano (Italy). All have taken several courses in object-oriented programming, software engineering, and distributed systems. Because of their background, they well represent the skills of a junior programmer. We organize the students in twelve groups of two students each, balancing competences based on the score the students obtained from the relevant courses. We assign every group a system among AR.Drone SDK, ROS, VOLTRONLIB, and VOLTRON. We give separate hands-on tutorials, of roughly two hours, about every such system. We also teach the students how to use the testbed we built to measure the run-time overhead, as described in Sec. 7.5. The students use the testbed to test their programs.

Throughout three half-day sessions, every group is given three increasingly complex programming exercises derived from the test applications described in Sec. 7.1. The first exercise is a relaxed version of the application that does not include adaptive sampling or time constraints. The second exercise includes the adaptive sensing aspects. The third exercise comprises the complete application. For example, ARCH-1 is the ARCH application without adapting the grid step based on aberration measurements and without requiring to check if the scan is completed in 10 minutes. ARCH-2 includes grid step adaptation. ARCH-3 encompasses both grid step adaptation and a time constraint, and equals ARCH.

At the end of the half-day, we check what every group produced and determine which groups successfully completed the exercises. To gather deeper insights, we also instruct the build environments to collect a snapshot of each group's code with every successful compilation process, and ask the students to fill an online questionnaire at the end of the whole study.

Results. Fig. 10 shows the outcome for ARCH, and reveals that only by using VOLTRON all groups are able to correctly complete all programming exercises. The results for PM_{10} and PURSUE are similar.

The reasons why the groups could not complete the exercises with systems other than VOLTRON are several. For example, we observe that groups using AR.Drone SDK do not complete the programming exercises because of lack of time. It simply takes too long for these groups to obtain a fully-functional solution for the simplest exercise and too little remains for the remaining exercises. The groups either turn in incomplete or buggy solutions. When explicitly asked through the questionnaire *"What is the most difficult aspect of programming coordinating drones using the system you were assigned?"*, the groups using AR.Drone SDK often complain that the learning curve is very steep and it is hard to implement even the simplest functionality.

Using ROS, the groups complete most of the exercises but do not complete them correctly. Those who try to build global knowledge out of the node-local states often fail to obtain consistent snapshots, and hence have drones make decisions based on inconsistent information. Those who try and apply a fully decentralized solution often do not consider some of the parallel executions, and fail to take care of the corresponding issues. We ask the groups *"Based on your programming experience in other fields, what would you compare the system you were assigned to?"*. One of the groups using ROS replies: *"It feels like programming an enterprise system using only C and sockets"*.

Finally, using VOLTRONLIB, all groups complete all programming exercises but the implementations are often defective due to subtle synchronization issues in handling concurrent tasks. When we ask in the questionnaire *"What would you try and improve, and how, of the programming system you were assigned?"*, all replies of groups using VOLTRONLIB point, in a way or the other, to concurrency handling. However, no one makes a concrete proposal about how to improve this aspect, indicating that the issue might not be too trivial.

Overall, the result indicate that: *i)* team-level programming greatly facilitates the programming of adaptive sensing multi-drone applications, as demonstrated by comparing the results of VOLTRONLIB and VOLTRON against AR.Drone SDK and ROS; and *ii)* the choice to implement the abstraction as a language extension helps programmers deal with concurrency issues between the drone operations in the field and data processing.

7.4 Software Metrics

We leverage established software metrics [26] to corroborate the results of the user study.

Metrics. When using an object-oriented paradigm, the number of *lines of code* (LOC), the number of *class attributes*, the number of *methods*, and the number of *method invocations* are generally considered as indications of a program's complexity [26]. It is also observed that complexity is a function of the number of *states* in which a class can find itself [26], where a state is any possible distinct assignment of values to a class' attributes during any possible execution. To carry out the latter analysis, we use our C++ prototype and SATABS [14], a verification tool for C/C++ programs.

Application	System	LOC	Per-class attributes (avg)	Per-class methods (avg)	Method invocations	Per-class states (avg)
ARCH	*ROS*	2354	10.9	9.3	181	≃32K
	AR.Drone SDK	3001	11.8	12.3	178	≃39K
	VOLTRONLIB	1409	6.1	5.3	64	≃9K
	VOLTRON	854	4.9	4.3	43	≃2K
PM_{10}	*ROS*	2311	9.7	12.1	101	≃30K
	AR.Drone SDK	2801	11.2	13.3	131	≃35K
	VOLTRONLIB	1385	6.7	8.1	61	≃8K
	VOLTRON	908	4.9	5.3	34	≃2K
PURSUE	*ROS*	3241	12.8	13.4	245	≃35K
	AR.Drone SDK	4512	12.9	13.3	298	≃43K
	VOLTRONLIB	1356	10.2	9.7	103	≃11K
	VOLTRON	1018	6.2	5.3	54	≃3K

Figure 11. VOLTRON simplifies implementing multi-drone active-sensing applications compared to VOLTRONLIB, ROS, and AR.Drone SDK.

SATABS exhaustively verifies C/C++ programs against user-provided assertions. Using a specific configuration, if the checking procedure terminates, SATABS returns the number of different program states it explores.

Results. Fig. 11 illustrates how both VOLTRON and VOLTRONLIB enable a reduction in all metrics, demonstrating that, in the applications we consider, team-level programming pays off even in the most straightforward instantiation. Directly comparing VOLTRON with VOLTRONLIB, we argue that the improvements of the former in LOC is due to the spatial semantics of variables and the system of futures and promises, illustrated in Sec. 4. The former spares the need to code the mapping between sensor values and the corresponding locations, whereas the latter greatly simplifies concurrency handling. The significant penalty of AR.Drone SDK over ROS in LOC is caused by the lack of support for inter-drone coordination in the former, hence demanding a great deal of manual coding of distributed functionality. This is in line with the results of the user study discussed earlier, where the groups using AR.Drone SDK often ran out of time.

Besides the LOC, the design of the application also appears simplified using VOLTRON. The individual classes include fewer attributes and fewer methods. Moreover, using VOLTRON the code becomes less entangled: fewer method invocations bind different classes. As a result, the code is likely easier to understand and to debug [26]. The average number of per-class states returned by SATABS also shows drastic improvements in favor of VOLTRON. Because of team-level programming and dedicated language extensions, the processing within the single classes is simplified.

7.5 Run-time Overhead

Accurately measuring system overhead in drone sensor networks is challenging in both simulation and in real-world experiments. To this end, we build a dedicated testbed based on ARM's developer tools [3] and the QEMU emulator [41]. The overhead measured will not be identical that observed in the field, but the results should be indicative because our setting is representative of current drone technology and actually models the most resource-constrained platforms.

Testbed. We employ the ARM Versatile Baseboard within QEMU to emulate the Cortex A8 processor. Using QEMU, we create a virtual machine running the same Linux image as the AR.Drone. We do not emulate the additional embedded board in charge of the virtual synchrony layer, which already proved to run efficiently in networks of hundreds of nodes [18]. We replace the virtual synchrony layer with a stub that implements a "perfect" virtual synchrony, that is, the total ordering follows the wall clock.

We implement dummy stubs for the AR.Drone navigation sensors and propellers' controls. We use a stub for the GPS sensor to play synthetic position traces generated according to where the drones are supposed to move. For simplicity, we let an AR.Drone always move along straight lines and at constant speed, thereby avoiding the need to generate synthetic traces for the navigation sensors since no corrections to the routes are ever needed. Thus, the flight control loop, although still running, bears no influence.

This setup allows us to create scenarios with up to 100 AR.Drones using 10 PCs connected via LAN, each running 10 instances of QEMU. The experiments are driven by one of the PCs, which generates random traces of sensor data that drive the execution. The same PC acts as the ground-station in centralized executions. Each application runs for about 25 minutes, corresponding to the maximum flight time of our custom AR.Drones. We repeat each run at least 100 times with different random traces of sensor data.

Metrics. *Memory consumption* (RAM) and *CPU utilization* on the drones may increase because the translation and the abstract machine add generic functionality that might be avoided in a hand-written implementation. Both figures determine the minimum system requirements. The generic functionality may also cause additional *network traffic* that an implementation using ROS or AR.Drone SDK may save. This figure affects a system's scalability, as excessive traffic may hog the network. Note that, since our GPS stub gives the location to a drone, these measures mainly depend on the *amount* of data to process. In other words, using random values does not affect the results.

To measure these quantities, we employ a SerialICE [41] connection to obtain detailed logs of CPU operation and memory consumption. We measure network traffic by sniffing exchanged packets. We repeat these measures for ROS, AR.Drone SDK, and VOLTRON running both centrally and in a state-replicated manner. We do not measure the VOLTRONLIB overhead because the run-time system is the same as VOLTRON. The results for state-replicated executions are to be considered merely as an indication of whether that execution model is at all feasible, as the numbers are not directly comparable with centralized executions. Every result refers to the performance of the single drone.

Results. Fig. 12 indicates that VOLTRON does introduce an overhead, but that the resource requirements are well within the capacity of current platforms.

Regardless of the programming system, most of the resources are *not* consumed by the application logic. As the bottom row in Fig. 12 demonstrates, merely keeping the AR.Drone at a fixed position doing nothing takes a significant amount of resources due to the tight flight control loop, which makes the drone hover by "locking" to features in the

Application	System	Memory [Mb] (avg/peak)	CPU utilization (avg)	Network traffic (pkts/min)
ARCH	*ROS*	86.7/88.6	39.9%	78.9
	AR.Drone SDK	86.2/88.2	41.1%	143.4
	VOLTRON *(centralized)*	93.7/92.7	49.6%	165.8
	VOLTRON *(state-replicated)*	95.1/93.3	62.1%	N/A
PM_{10}	*ROS*	86,8/89.2	42.5%	70.1
	AR.Drone SDK	85.2/87.3	43.7%	101.3
	VOLTRON *(centralized)*	89.7/93.5	52.1%	123.4
	VOLTRON *(state-replicated)*	91.3/96.7	65.1%	N/A
PURSUE	*ROS*	87.1/89.6	37.7%	74.7
	AR.Drone SDK	87.1/88.5	35.7%	83.2
	VOLTRON *(centralized)*	95.6/100.5	56.2%	154.6
	VOLTRON *(state-replicated)*	96.2/100.6	71.7%	N/A
	Idle (hovering)	84.3/84.3	30.5%	0

Figure 12. With 100 emulated drones, VOLTRON introduces a modest run-time overhead.

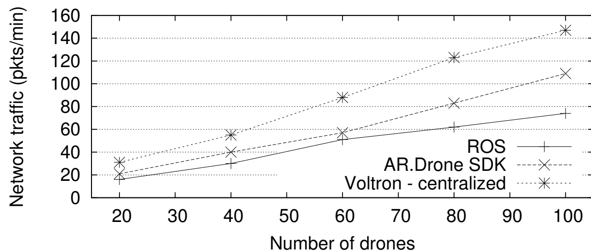

Figure 13. Average network traffic across all applications against number of drones.

video feed from the vertical camera.

The measures for network traffic in Fig. 12 indicate that the throughput is well below the limits of current WiFi networks, such as those aboard modern drones [1]. The worse performance of AR.Drone SDK compared to ROS is caused by the lack of multicast communication support, so coordination among drones requires 1-to-1 message exchanges using AR.Drone SDK. In VOLTRON, additional traffic is caused by bookkeeping information. In state-replicated executions, communication among drones occurs through the (multi-hop) 802.15.4 network enabled by the virtual synchrony layer [18].

Memory consumption and CPU utilization are independent of the number of drones, at least for centralized executions where the nodes only interact with the ground-station. Similarly, network traffic grows about linearly regardless of the programming system, as shown in Fig. 13. These observations, together with the absolute values in Fig. 12, suggest that VOLTRON would likely scale beyond the 100 drones in these experiments before reaching resource limits, for these applications.

8 Deployment Assessment

We perform a preliminary evaluation of the accuracy of the orthophotos produced by our system at the *Domus dei Putti Danzanti*. For ground truth, we use distance measurements taken using professional LIDAR equipment between five points uniformly chosen. We evaluate the orthophotos obtained using all seven drones, that is, the only orthophotos that could be obtained while meeting the aberration requirements and quickly enough for the scene not to change. The same five points were identified in the orthophoto itself us-

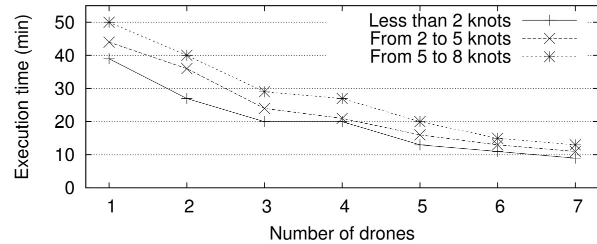

Figure 14. Wind conditions bear an appreciable effect on execution times.

ing visual makers on the ground. Remarkably, the distances between the points on the orthophoto are within a 3.3% error margin of the LIDAR measurements.

Dually, the other key performance metric for the ARCH application is the time required to obtain the orthophoto. At the *Domus dei Putti Danzanti*, current archaeological practices involve a great deal of manual labor: archaeologists use measuring tape and hand-made drawings, along with pictures taken with a hand-held camera. Two graduate students are allocated full-time to these procedures which, needless to say, are immensely time consuming and error-prone. If nothing else, one cannot just freely walk around for fear that one's foot may irreparably damage something still covered by a thin layer of dust.

In about five months, we almost completely automated these procedures using our centralized Java-based VOLTRON prototype, a laptop PC as ground-station together with an amplified WiFi directional antenna, and seven custom AR.Drones. A video showing the drones in action at the *Domus dei Putti Danzanti*, against particularly windy conditions, is found at `http://youtu.be/PPDGO-jc0It`.

However, aerial drones are not completely labor free, even if they fly autonomously. We learned the hard way that the time saved by using multiple drones can quickly evaporate without proper mechanical maintenance. Archaeological sites are particularly dusty environments and, as dust enters a drone's gears and shafts, their efficiency quickly drops until the drone fails to operate. We eventually identified a suitable maintenance schedule but, in the meantime, we broke four motors, eleven propellers, and uncountable gears.

Additionally, execution times can be highly variable, as shown in Fig. 8, due to unpredictable wind effects. For a better understanding, we deployed a digital anemometer in the middle of the site. Fig. 14 shows the execution times depending on the average wind conditions during an experiment. As the wind increases, the drones invest more effort in counteracting its action. This suggests that, in the absence of other requirements, the drone operation may just be postponed if the current wind conditions are unfavorable.

We also found that application-level performance of real-world drone applications depend greatly on hardware design choices. As an example, the choice of camera creates a trade-off between image quality and weight/size. The former affects the grid size requirements while the latter affects energy consumption and hence the flight times because of the additional payload. We eventually settled on a Polaroid XS10 action camera, which provides 3 megapixel pictures at 120° view angle—sufficient for the archaeologists' needs—

in only 24 grams. It is also small enough to to be hooked under the AR.Drone with no special mounting, and hence no further weight. In addition, the low-cost GPS receiver we chose had accuracy issues even with clear-sky conditions at the site. More expensive GPS receivers could have produced better accuracy and therefore could have reduced image aberration and therefore scan time.

All things considered, our system was able to reduce an entire day of work for two graduate students down to about half an hour of autonomous drone operation, plus two hours of image post-processing for a single person and one hour per week for drone maintenance.

9 Conclusion

We designed the team-level programming model to ease the implementation of active sensing applications using multiple coordinating autonomous drones. We implemented team-level programming in VOLTRON, a system that extends an existing language with a notion of abstract drone and by means of variables enriched with spatial and temporal semantics. We assessed VOLTRON based on a real prototype we built, supporting centralized and state-replicated executions on custom AR.Drone 2.0 devices. By means of experiments in a real deployment, we studied the improvements in execution time by using multiple drones. Based on this and two more representative applications, we identified through user studies and software metrics significant advantages in programming multi-drone active sensing applications using VOLTRON compared to alternative solutions, at the price of a modest increase in resource consumption. We concluded by reporting lessons from the real deployment we carried out.

10 Acknowledgments

We wish to thank Federica Fontana and Emanuela Murgia at the University of Trieste for the fruitful collaboration at the Aquileia deployment, as well as our paper shepherd Tarek Abdelzaher and the anonymous reviewers for their insightful comments. The work was supported by the EU programme IDEAS-ERC, project EU-227977 SMScom, and by SSF, the Swedish Foundation for Strategic Research.

11 References

[1] 3D Robotics. UAV Technology. goo.gl/sBoH6.
[2] Parrot AR.Drone SDK. goo.gl/wLaur.
[3] ARM Developer Tools. goo.gl/MijD2y.
[4] M. P. Ashley-Rollman et al. A language for large ensembles of independently executing nodes. In *Collections on Logic Programming*. 2009.
[5] Autonomy Lab - Simon Fraser University. AR.Drone ROS driver. goo.gl/i5Us5D.
[6] J. Bachrach et al. Composable continuous-space programs for robotic swarms. *Neural Computation and Applications*, 2010.
[7] C. H. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proc. of the Symp. on Artificial Intelligence and Programming Languages*, 1977.
[8] BBC News. Disaster drones: How robot teams can help in a crisis. goo.gl/6efliV.
[9] T. Bektas. The multiple traveling salesman problem: An overview of formulations and solution procedures. *International Journal of Management Science*, 2006.
[10] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proc. of SOSP*, 1987.
[11] M. Brambilla et al. Swarm robotics: A review from the swarm engineering perspective. *Swarm Intelligence*, 2013.
[12] A. Bürkle. Collaborating miniature drones for surveillance and reconnaissance. In *Europe Security+ Defence*, 2009.
[13] C. Anderson. How I accidentally kickstarted the domestic drone boom. goo.gl/SPOIR.
[14] E. Clarke et al. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. of TACAS*, 2005.
[15] CNN.com. Your personal $849 underwater drone. goo.gl/m1JRuD.
[16] DailyMail. Meet Ray: Düsseldorf airport's autonomous robot car parking concierge. goo.gl/6E1lTY.
[17] K. Dantu et al. Programming micro-aerial vehicle swarms with Karma. In *Proc. of SENSYS*, 2011.
[18] F. Ferrari et al. Virtual synchrony guarantees for cyber-physical systems. In *Proc. of SRDS*, 2013.
[19] F. Fontana and E. Murgia. The "Domus dei Putti Danzanti" along the Gemina route: Element of the decorative apparatus. In *Proc. of the Conf. on Roman Architecture*, 2012. In Italian.
[20] D. Friedman and D. Wise. The impact of applicative programming on multiprocessing. In *Proc. of the Conf. on Parallel Processing*, 1976.
[21] B. Gerkey et al. The Player/Stage Project. In *Proc. of ICAR*, 2003.
[22] iRobot. Create Programmable Robot. goo.gl/bJhrMR.
[23] A. Kansal et al. Building a sensor network of mobile phones. In *Proc. of IPSN*, 2007.
[24] J. Kramer and M. Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2), 2007.
[25] F. Lai et al. CSense: A stream-processing toolkit for robust and high-rate mobile sensing applications. In *Proc. of IPSN*, 2014.
[26] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.
[27] U. Lee et al. Mobeyes: Smart mobs for urban monitoring with a vehicular sensor network. *Wireless Commun.*, 2006.
[28] P. U. Lima and L. M. Custodio. *Multi-Robot Systems*. Springer, 2005.
[29] T. Liu et al. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. of MOBISYS*, 2004.
[30] N. Michael et al. Cooperative manipulation and transportation with aerial robots. *Autonomous Robots*, 30(1):73–86, 2011.
[31] L. Mottola and G.P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comp. Surv.*, 2011.
[32] N. Stefanos. Orthophoto producer. goo.gl/FTjcUa.
[33] F. Nex and F. Remonding. UAV for 3D mapping applications: A review. *Applied Geomatics*, 2003.
[34] Y. Ni et al. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. of PLDI*, 2005.
[35] E. Nocerino et al. Accuracy and block deformation analysis in automatic UAV and terrestrial photogrammetry: Lessons learnt. In *CIPA Symposium*, 2013.
[36] Paparazzi Autopilot Project. goo.gl/bwwXjX.
[37] Parrot. AR.Drone 2.0. goo.gl/n292Bw.
[38] B. Pásztor et al. Selective reprogramming of mobile sensor networks through social community detection. In *Proc. of EWSN*, 2010.
[39] PixHawk.org. PX4 autopilot. goo.gl/wU4fmk.
[40] A. Purohit et al. Sensorfly: Controlled-mobile sensing platform for indoor emergency response applications. In *Proc. of IPSN*, 2011.
[41] QEMU Machine Emulator. goo.gl/AVPvRP.
[42] M. Quigley et al. ROS: An open-source robot operating system. *ICRA Workshop on Open Source Software*, 2009.
[43] R. Ritz et al. Cooperative quadrocopter ball throwing and catching. In *Proc. of IROS*, 2012.
[44] T. Stirling et al. Energy efficient swarm deployment for search in unknown environments. In *Proc. of ANTS*, 2010.
[45] U.S. Environmental Protection Agency. Air Pollutants. goo.gl/stvh8.
[46] J. Villasenor. Observations from above: Unmanned Aircraft Systems. *Harvard Journal of Law and Public Policy*, 2012.
[47] Z. Zhang. Routing in intermittently-connected mobile ad-hoc networks and delay-tolerant networks: Overview and challenges. *Commun. Surveys Tuts.*, 2006.