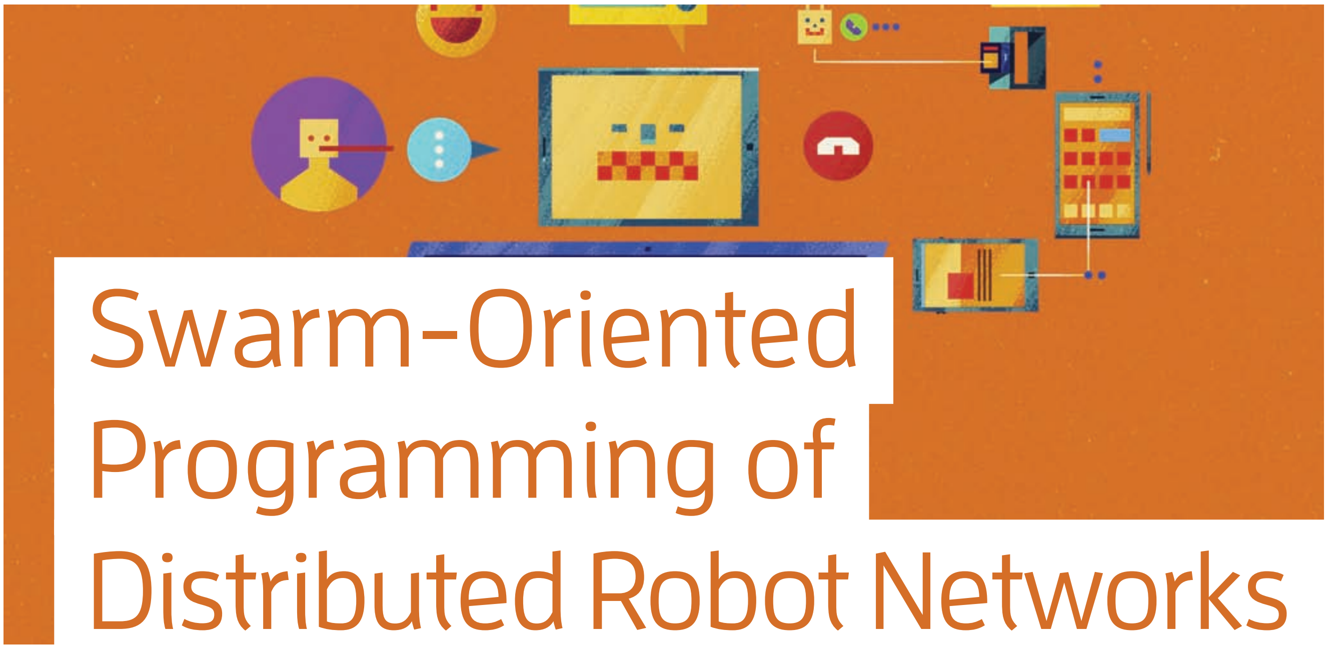


Swarm-Oriented Programming of Distributed Robot Networks

Carlo Pinciroli, Worcester Polytechnic Institute
Giovanni Beltrame, École Polytechnique de Montréal

One challenge of programming soon-to-be-common large robotic teams is the definition of programming primitives that generate reusable and predictable behaviors. A new language construct allows developers to categorize robots using custom conditions and assign tasks to the groups created.

From delivery drones to self-driving cars, robots will become increasingly pervasive and interconnected in society. Eventually, this robotics revolution will fundamentally change the type of devices available to the general public. Unlike the current concept of "smart devices," which can sense and compute autonomously, robots can integrate sensing, computation, and actuation in the physical world. Such autonomous systems will be instrumental for safety-critical applications, including search-and-rescue operations, industrial and agricultural inspection, surveillance, mining, construction, self-driving manned vehicles, and surgery. The large-scale and dynamic nature of these applications requires multiple robots to interact effectively. We envision a world in which networked robotic devices will coordinate in teams to achieve complex tasks.

One of the current challenges in this field is to define suitable programming primitives that generate well-structured, reusable, and predictable team behaviors. To help achieve this goal, we developed the `swarm` language construct, which overcomes the all-or-nothing choice between focusing either on individual robots or the team of robots as a whole. In our meet-in-the-middle approach, the `swarm` construct allows developers to categorize robots with respect to certain conditions and assign tasks to robots that belong to a certain swarm. In this article, we show how the programs produced using this construct are structured into well-defined, short, and reusable functions.

The `swarm` construct is one of the novel aspects of the Buzz programming language (the.swarming.buzz),[1] which we recently created. Because robot swarms belong to the wider class of autonomous, distributed devices, the implications of our proposed approach are not limited to the robotics world but can be applied to any distributed, multidevice, autonomous system such as smart sensor networks or distributed applications for mobile devices.

TRENDS IN SWARM PROGRAMMING

The research community has increasingly focused on the problem of coordinating mid- and large-scale teams of robots, or *swarms*.[2] Current approaches are generally either *centralized methods*, wherein a single planner collects information and dictates the robots' actions, or *decentralized methods*, wherein every robot acts accordingly to its limited on-board information. Decentralized methods promise better robustness to failure and better parallelism, but they demand more complex designs than centralized methods.

The complexity of designing decentralized robotics systems hinges on the fact that functional requirements are expressed at the swarm level, while actions must be executed at the individual level. This is often referred to as the *global-to-local problem*. During development, the challenge is to select a suitable abstraction level to express swarm behaviors. The current approaches to swarm programming can be categorized by the granularity of their abstraction level.

Broadly speaking, swarm programming approaches commonly fall into two categories: bottom-up or top-down. The bottom-up approach is an adaptation of a single-robot behavior design in which the developer focuses on individual robot behaviors. The development process proceeds by trial and error, progressively refining individual behaviors until swarm-level requirements are satisfied. Programming languages such as C, C++, and Python, often integrated with frameworks such as the Robot Operating System (ROS; www.ros.org)[3] or DroneKit (www.dronekit.io), are common choices to pursue this approach. Top-down development, on the other hand, stems from research in sensor networks[4] and spatial computing,[5] and focuses on the swarm as a whole. A number of domain-specific languages belong to this category, such as Proto/Protelis[6,7] (for aggregate programming) and Meld[8] (for self-assembling robots).

Bottom-up development offers a high level of system control but requires the developer to supply an overwhelming amount of detail,[5] including communication protocols, low-level control, and (with C/C++) even memory management. Top-down development, on the other hand, allows the developer to concentrate on swarm-related algorithms, sacrificing design flexibility on many aspects of the system, such as actuation.[5] We believe this all-or-nothing choice between bottom-up and top-down programming is the result of a lack of interaction between the distributed robotics (swarm robotics, in particular) and software engineering domains. Recent efforts in robotics software development, most notably the upcoming second version of ROS, will allow advanced software engineering techniques to find their way into distributed robotics scenarios.

We believe that it is possible to avoid the choice between bottom-up and top-down programming in robotics. Using the `swarm` construct, developers can tag robots so they respect specific conditions and assign tasks to robots as a function of their tags. Tagging robots conditionally offers a mixed level of abstraction that enables developers to express complex swarm algorithms in a comfortable and concise manner. Robots can also have multiple tags, which further increases the flexibility of task assignments.

BUZZ PROGRAMMING LANGUAGE

The main goal of the Buzz programming language[1] is to provide developers with a small but powerful set of primitives that enables the expression of complex swarm-robotics algorithms. Buzz is also designed as an extension language. Rather than replacing the software stack already present on a robot, Buzz is designed to integrate with it seamlessly, exposing only the relevant aspects of a robot's API. This design choice makes it possible to use Buzz with virtually any type of robot and to add application-specific primitives to the language. In a heterogeneous robot swarm that uses different software frameworks (such as ROS for ground-based robots and DroneKit for quad-rotors), a Buzz layer effectively eliminates the low-level differences and enables code reuse and performance comparisons.

From a computational point of view, Buzz sees a swarm as a discrete collection of interacting robots. Each robot runs an instance of the Buzz Virtual Machine (BVM), on which a Buzz script executes. The current version of the Buzz runtime assumes that the same script runs on every robot.

The robots can be heterogeneous in their capabilities, but they must be able to exchange information and detect one another. Buzz is based on a form of inter-robot communication called *situated communication*. This communication modality is based on a local message broadcast that is "situated" because a robot, upon receiving a message, can detect the sender's position with respect to its own frame of reference. Situated communication forms the base of a large number of swarm algorithms, including pattern formation, task allocation, aggregation, and exploration.

At first look, the Buzz syntax resembles JavaScript, Python, and Lua. This choice was intentional to ensure a short learning curve for newcomers. The Buzz primitives include classical constructs—such as variable and function definitions, branches, and loops—as well as higher-level constructs designed for spatial and network coordination. The `neighbors` construct, for instance, is a data structure that contains information about the other robots within communication range. Through this construct, it is possible to broadcast messages, aggregate local information, filter robots and their associated information, and spatially interact with nearby robots. To propagate information globally across the swarm, Buzz offers the `stigmergy` construct,[9] a lightweight distributed hash table that is based on local broadcast and is resistant to severe message loss.

SWARM CONSTRUCT

The Buzz `swarm` construct lets developers tag a set of robots according to a certain condition. Although the swarm syntax is simple, it allows for a wide range of possible applications, making the `swarm` construct a powerful programming tool.

Programming interface

A swarm is created using a call to the `swarm.create()` method. This method takes a unique numerical identifier as a parameter, which is internally used by the BVM to manage swarm membership: `s = swarm.create(1)`.

A newly created swarm is empty. Robots can be assigned to the new swarm using the `swarm.select()` or `swarm.join()` methods. The `swarm.select()` method lets developers express a condition evaluated individually by every robot at runtime. If the given condition is satisfied, the robot joins the swarm. The `swarm.join()` method lets a robot join a swarm unconditionally:

```
# If the robot identifier is even,
# the robot joins the swarm
s.select(id % 2 == 0)
# Every robot joins the swarm
# unconditionally
s.join()
```

The logic of leaving a swarm is analogous and achieved through the `swarm.unselect()` and `swarm.leave()` methods.

A swarm lambda is a function assigned to a specific swarm for execution. When a swarm lambda is specified, every robot in the swarm executes it:

```
# All the drones in swarm s take off
s.exec(function() { takeoff() })
```

Internally, the lambda is executed as a swarm function call. This call modality differs from classical function calls in that the current swarm ID is pushed onto a dedicated swarm stack. Upon return from a swarm function call, the swarm stack is popped. The BVM manages the swarm stack to keep track of nested swarm function calls. When the swarm stack is not empty, the `swarm.id()` method is defined. If called without arguments, the `swarm.id()` method returns the swarm ID at the top of the swarm stack (that is, the current swarm ID); if passed an integer argument $n > 0$, it returns the nth element in the swarm stack. Other Buzz primitives, such as `neighbors` and `stigmergy`, also use the swarm stack to limit the scope of their operations to the robots in the stack-top swarm.

Swarms can be considered sets of robots. Therefore, it is possible to give them standard set operations, such as intersection, union, difference, and negation. Each of these operations results in the creation of a new swarm:

```
# We assume swarms s1 and s2 have
# been created and filled
# Intersection of s1 and s2 (robots
# belonging to both s1 and s2)
# Parameters: the swarm id, and the
# list of swarms to intersect
i = swarm.intersection(100, s1, s2)
# Union of s1 and s2 (robots
# belonging to s1 and/or s2)
# Parameters: the swarm id, and the
# list of swarms to merge
u = swarm.union(101, s1, s2)
# Difference of s1 and s2 (robots
# belonging to s1 and not s2)
# Parameters: the swarm id, and the
# list of swarms to consider
d = swarm.difference(102, s1, s2)
# Negation of s1 (robots not
# belonging to s1)
n = swarm.others(103)
```

Through the combined use of `swarm.select()`, `swarm.unselect()`, and the set operations, it is possible to express complex dynamics that evolve over time. Using the `swarm.select()` method, we can create swarms according to conditions that are satisfied at the specific time they are evaluated. The condition that defines a swarm created using this method might become false at some point, but this will not affect the swarm composition. If the condition is intended as a membership requirement, however, the `swarm.unselect()` method offers a simple way to enforce this requirement. In contrast, set operations can be used to manipulate swarms according to their current composition, rather than on the condition they satisfied at any moment in time. In other words, set operations make it possible to select robots that are (or are not) part of one or more swarms, effectively treating swarm membership as a sort of tag applied to a robot. As we will show here, a careful definition of the swarms lets us structure a complex swarm algorithm into a well-defined set of small, highly reusable functions.

Low-level implementation

The BVM transparently manages swarm information. Essentially, each robot maintains two data structures about swarms: the first structure notes the swarms to which the robot belongs (its memberships), and the second stores data regarding neighboring robots. The latter structure, called the *neighbor membership table*, is critical to the correctness of a Buzz script.

Every time a `swarm.create()` method is executed, the BVM stores the identifier of the created swarm in a dedicated hash table, along with a flag encoding whether the robot is a member of the swarm (`1`) or not (`0`). When a robot is added to a swarm, the BVM sets the flag corresponding to the swarm to 1 and queues a message `<SWARM_JOIN, robot_id, swarm_id>`. Analogously, when a robot leaves a swarm, the BVM sets the corresponding flag to 0 and queues a message `<SWARM_LEAVE, robot_id, swarm_id>`. Because leaving and joining swarms are not particularly frequent operations, and a robot's neighborhood constantly changes as it moves, it is likely for a robot to encounter a neighbor for which no information is available. To keep every robot's information up to date, the BVM periodically queues a message `<SWARM_LIST, robot_id, swarm_id_list>` that contains the complete list of swarms to which the robot belongs. When configuring the BVM installed on a robot, developers can set the frequency of this message.

The neighbor membership table is a hash map indexed by robot IDs. Each element of the hash map is a (`swarm_id_list`, `age`) pair where `swarm_id_list` corresponds with the list of swarms to which the robot is a member, and `age` is a counter of the time steps since the last reception of a swarm update. Upon receiving a swarm-related message (such as `swarm_join`, `swarm_leave`, or `swarm_list`), the BVM updates the information on a robot accordingly and zeroes the age of the entry. This counter is employed to delete information on robots for which no message has been received in a predefined period. When the counter exceeds a certain threshold (set by developers when the BVM is configured), the information on the corresponding robot is removed from the structure. This simple mechanism lets the robots focus memory storage on active, nearby robots, avoiding a waste of resources on unnecessary robots, such as those that are damaged or out of range. In addition, this mechanism prevents excessive memory usage when the swarm size increases.

To minimize the bandwidth required for swarm management, the BVM performs a number of optimizations on the queue of swarm-related outbound messages. These optimizations involve identifying the minimum number of messages necessary to send in order to communicate the current state of a robot's swarm membership. (The details on this aspect exceed the scope of this article, but are available in earlier work.[1])

BEYOND GLOBAL VERSUS LOCAL GRANULARITY

The semantics of the `swarm` construct overcome the global-to-local granularity problem by allowing developers to dynamically define the set of robots that should execute a certain portion of code. We provide four approaches to defining swarm behaviors, showing how the `swarm` construct can be used to express them.

Structure: swarm-level finite state machine

In a seminal paper, Alcherio Martinoli and his colleagues showed how swarm dynamics can be described using a finite state machine (FSM) in which each state is marked with the fraction of robots currently in that state.[10] The swarm dynamics can then be described as a flow of robots that transition from state to state. A state typically corresponds to an individual robot behavior under execution. By describing the dynamics of a swarm through an FSM, developers can focus on the sequence of behaviors to execute and on the transitions among them, rather than on the individual robots.

The `swarm` construct captures this concept naturally. Each FSM state is mapped to a dedicated swarm, and the transition logic is realized through calls to `swarm.unselect()` or `swarm.leave()`, and `swarm.select()` or `swarm.join()`. Figure 1 shows an example in which robots start in Behavior 1. Upon executing a behavior, a robot switches to the Idle state, waiting for the other robots to finish. As soon as all the robots are idle, they switch to Behavior 2. The code also shows how the `stigmergy` construct can be used to implement a simple barrier to wait for all robots to be done with a behavior: each robot stores its ID in the hash

```
# Number of robots in the swarm, assumed known and fixed for simplicity
SWARM_SIZE = 100
# A numeric id for the barrier stigmergy
BARRIER_VSTIG_ID = 1
# Function to wait for everybody to be ready
function barrier_wait() {
  if(barrier.size() < SWARM_SIZE) barrier.get(id);
}
# Swarm states
STATE_BHVR1 = 1
STATE_IDLE = 2
STATE_BHVR2 = 3
# This function is executed at initialization.
function init() {
  # Create swarms for each state
  bhvr1 = swarm.create(STATE_BHVR1)
  idle  = swarm.create(STATE_IDLE)
  bhvr2 = swarm.create(STATE_BHVR2)
  # Initially every robot is doing behavior 1
  bhvr1.join()
}
# Logic of behavior 1
function do_bhvr1() {
  var done = 0
  ...
  # We assume 'done' contains 1 if the behavior is finished
  if(done) {
    # Switch to idle state
    bhvr1.leave()
    idle.join()
    # Create a barrier using a stigmergy structure
    barrier = stigmergy.create(BARRIER_VSTIG)
    barrier.put(id, 1)
  }
}
# Logic of idle state
function do_idle() {
  # Is everybody done?
  if(barrier.size() == SWARM_SIZE) {
    # Switch to behavior 2
    idle.leave()
    bhvr2.join()
  }
}
# Logic of behavior 2
function do_bhvr2() {
  ...
}
# This function is executed at each time step
function step() {
  bhvr1.exec(dobhvr1)
  idle.exec(doidle)
  bhvr2.exec(dobhvr2)
}
```

FIGURE 1. Example of a finite state machine implemented in the Buzz programming language. This code also shows how to use the `stigmergy` construct to implement a simple barrier.

table, and when the size of the hash table matches the swarm's size, the barrier is lifted.

Coordination: distributed task allocation

Two recent works pioneered a new approach to distributed robot programming. Rather than concentrating on the robots, the Karma[11] and Voltron[12] languages focus on the tasks that the swarm must perform. In other words, the developer specifies only the logic associated with a task, ignoring which robot will eventually execute it. The tasks are geographically distributed and assumed executable at any time. A dedicated runtime framework assigns robots to tasks dynamically and transparently from the developers' point of view. Both Karma and Voltron target single-robot single-task sensory scenarios, in which the robots move and observe the environment, but the two languages differ in the way the tasks are coordinated.

The `swarm` construct offers a way to obtain a similar result and to maintain full control over the process. The core idea is to structure the Buzz script in two macro-components: the dispatcher and the tasks. The dispatcher's role is to keep track of task completion and to manage the robots' transitions from one task to another; each task corresponds to a dedicated swarm, and the logic associated with a task is a suitably defined Buzz function. Figure 2 shows a simple example of this approach.

The advantage of structuring the code in this way is that it decouples task dispatching and task logic, making it possible to modify, improve, and reuse a component. This is particularly useful because it fosters meaningful comparisons among distributed task-dispatching algorithms and allows for more ambitious research toward decentralized planning of interdependent tasks.

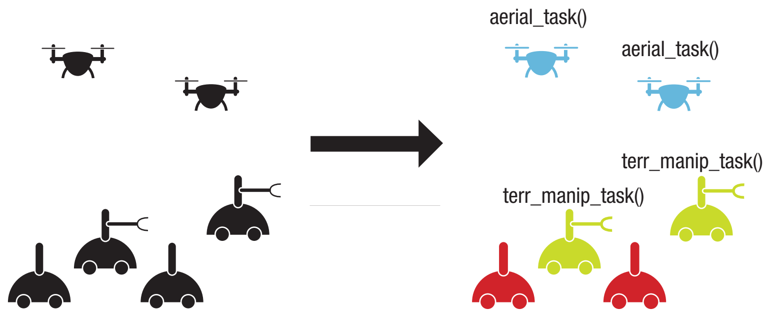
Heterogeneity: feature tagging

An important aspect of future robot swarms is that their composition will be highly heterogeneous. From a hardware point of view, the benefits of heterogeneity are two-pronged. First, there are lower costs associated with producing large quantities of specialized robots, rather than a small number of generic ones. Second, it will be possible to deploy task-specific robots over time, lowering the costs and complexity of swarm deployment. Other sources of heterogeneity might be information (only some robots need to be aware of a certain fact about the environment) and location (being in range to engage a target).

The `swarm` construct can be used to tag robots belonging to a certain category. Hardware heterogeneity is detectable through the presence or absence of specific language symbols. For instance, the BVM installed on a quad-rotor is likely to contain the definition of the function `fly_to()`, whereas a ground robot would have `set_wheel_speed()`. The developer can make specific choices regarding which symbols are present on which robot when designing the robots' BVM integration, and it is (at least partially) application specific. Figure 3 shows an example of swarm creation based on hardware heterogeneity in a swarm that consists of quad-rotors and two types of ground-based robots, one of which is equipped with a manipulator. Information heterogeneity can be captured by creating swarms with conditions related to specific variables. Analogously, location heterogeneity can be dealt with

```
# Total number of tasks to perform
TASKNUM = 2
# This function is executed at initialization
function init() {
  # Create a swarm for each task
  idle  = swarm.create(1)
  task1 = swarm.create(2)
  task2 = swarm.create(3)
  # Initially every robot is idle
  idle.join()
  # Create a stigmergy to store task completion status
  taskcomp = stigmergy.create(1)
  taskcomp.put(1, task1)
  taskcomp.put(2, task2)
}
# A simple task dispatcher that assigns to a robot the first
# available task
function task_dispatch() {
  # Check if more tasks are available
  if(taskcomp.size() > 0) {
    # Pick the first uncompleted task
    for(var i = 0; i < TASKNUM; i = i + 1) {
      if(taskcomp.get(i)) {
        # Switch to task
        taskcomp.get(i).join()
        # Remove task from pool
        taskcomp.put(i, nil)
      }
    }
  }
}
# Idle robot logic
function do_idle() {
  task_dispatch()
}
# Task 1 logic
function do_task1() {
  var done = 0
  # Logic of task 1
  ...
  if(done) {
    # Switch to idle
    task1.leave()
    idle.join()
  }
}
# Task 2 logic
function do_task2() {
  var done = 0
  # Logic of task 2
  ...
  if(done) {
    # Switch to idle
    task2.leave()
    idle.join()
  }
}
# This function is executed at each time step
function step() {
  idle.exec(do_idle)
  task1.exec(do_task1)
  task2.exec(do_task2)
}
```

FIGURE 2. A simple task dispatcher in Buzz. The code has two components: the task dispatcher (encoded in the `task_dispatch()` function) and the actual tasks. Each task is expressed through a swarm, which executes the corresponding logic encoded in the `do_task*()` functions. The `stigmergy` structure is used to keep track of the completed tasks.



```
# Group identifiers
AERIAL          = 1
TERRESTRIAL     = 2
MANIPULATORS    = 4
# Task for aerial robots
function aerial_task() { ... }
# Task for terrestrial manipulator robots
function terr_manip_task() { ... }
# Create swarm with robots possessing the ‘fly_to’ symbol
aerial = swarm.create(AERIAL)
aerial.select(fly_to)
# Create swarm with robots possessing the ‘set_wheel_speed’ symbol
terrestrial = swarm.create(TERRESTRIAL)
terrestrial.select(set_wheel_speed)
# Create swarm with robots possessing the ‘grip’ symbol
manipulators = swarm.create(MANIPULATORS)
manipulators.select(grip)
# Assign task to terrestrial manipulators
terr_manip = swarm.intersection(TERRESTRIAL + MANIPULATORS,
terrestrial, manipulators)
terr_manip.exec(aerial_task)
# Assign task to aerial robots
aerial.exec(terr_manip_task)
```

FIGURE 3. Using `swarm` variables, developers can create swarms and assign them tasks. In this example, three swarms are created based on hardware heterogeneity, and two of them are assigned tasks to execute.

by conditions that are dependent on sensor readings.

Coherence: team tagging

Large robot swarms are often divided into teams that must operate as a single entity. These teams typically achieve spatial coherence by aggregation, flocking, or self-assembly before moving toward the designated area for their mission. An important problem in this scenario is maintaining team coherence when two teams operate in close proximity. For instance, when two flocks meet, the robots must be able to recognize similar (kin) robots and maintain formation with them, while avoiding strangers.

The `swarm` construct offers a solution to this problem. As we explained earlier, when a swarm lambda is executed, a robot is aware of the swarms it belongs to. Some Buzz primitives, such as `neighbors` and `stigmergy`, internally use the swarm stack to limit their scope or to perform particular operations. Figure 4 shows an example in which two swarms interact. The robots that belong to the same swarm maintain a short distance from one another, while keeping a farther distance from strangers. The core of the algorithm is in the `neighbors.kin()` and `neighbors.nonkin()` methods. The first method returns a list of robots belonging to the same swarm, and the second returns a list of known robots that do not belong to the swarm.

VALIDATION

To assess the efficiency of the framework that supports the `swarm` construct, we conducted a series of experiments, both in simulation and with real robots. For the simulated experiments, we used the ARGoS multirobot simulator (www.argos-sim.info). For real-world assessment, we used a swarm of 10 Khepera IV robots (www.k-team.com/khepera-iv). Our experiments were designed to highlight the dynamics of the swarm coordination subsystems. For space reasons, we report only a part of the results here. The complete set of experiments and related videos, along with the code necessary to run them, is available at the.swarming.buzz/papers/IEEEComputer2016.

Experimental setup

We considered a scenario in which a swarm of N robots is deployed in an environment. Every experiment lasts a total of $T = 2$ minutes. We chose this duration because early experiments showed that the dynamics of the `swarm` construct stabilize in just a few seconds. Also, in the experiments involving motion, 2 minutes is sufficient for the robots to cover a reasonable portion of the arena. In our experiment, the robots created two swarms with IDs 1 and 2. All robots joined swarm 1 and communicated with their neighbors. After 60 seconds, all robots left swarm 1 and joined swarm 2.

An important parameter in our simulated experiments is the probability of message loss *P*. We studied three values for this probability: 0, 0.25, and 0.50. As a performance measure, we observed how long it took the robots to detect the change in swarm membership in their neighborhood. Specifically, we collected data on the neighbors observed by each robot and counted the number of correct and incorrect entries in each robot's neighbor membership table.

Scalability and topology dependency

Scalability and topology are important factors in multirobot coordination. To assess the impact of different topologies on the management of swarm-related information, we considered three scenarios in which the robots were initially placed in the environment and did not move. In the first scenario, the robots were deployed in a tight cluster where every robot could communicate with a large number of neighbors. In the second scenario, we considered the opposite case—a line topology where every robot had at most two direct neighbors. The third was an in-between scenario—a scale-free topology, in which the distribution of the number of robot neighbors followed a power law.

To assess scalability, we analyzed the system performance when the swarm size *N* was 10, 100, and 1,000 robots. We tested every individual parameter choice (topology, *N*, *P*) 30 times for a total of 810 experiments. Figures 5a and 5b illustrate the results obtained for the scale-free scenario, with a message

```
# Constants
TARGET_KIN     = 100.
EPSILON_KIN    = 100.
TARGET_NONKIN  = 300.
EPSILON_NONKIN = 250.
# Lennard-Jones interaction magnitude
function calc_lj(dist, target, epsilon) {
  return -(epsilon / dist) * ((target / dist)^4 - (target / dist)^2)
}
# Neighbor data to kin LJ interaction
function to_lj_kin(rid, data) {
  var lj = calc_lj(data.distance, TARGET_KIN, EPSILON_KIN)
  data.x = lj * math.cos(data.azimuth)
  data.y = lj * math.sin(data.azimuth)
  return data
}
# Neighbor data to non-kin LJ interaction
function to_lj_nonkin(rid, data) {
  var lj = calc_lj(data.distance, TARGET_NONKIN, EPSILON_NONKIN)
  data.x = lj * math.cos(data.azimuth)
  data.y = lj * math.sin(data.azimuth)
  return data
}
# Accumulator of neighbor LJ interactions
function vec2_sum(rid, data, accum) {
  accum.x = accum.x + data.x
  accum.y = accum.y + data.y
  return accum
}
# Actual flocking logic
function flock() {
  # Create accumulator
  var accum = { .x=0, .y=0}
  # Calculate accumulator
  accum = neighbors.kin().map(to_lj_kin).reduce(vec2_sum, accum)
  accum = neighbors.nonkin().map(to_lj_nonkin).reduce(vec2_sum, accum)
  if(neighbors.count() > 0) {
    accum.x = accum.x / neighbors.count()
    accum.y = accum.y / neighbors.count()
  }
  # Move according to vector
  goto(accum.x, accum.y)
}
```

FIGURE 4. An example Buzz script for defining spatial interaction among swarms. Using the `neighbors.kin()` and `neighbors.nonkin()` methods, it is possible to encode a logic that maintains a short distance between robots in the same swarm and a farther distance between robots in different swarms.

drop probability of 0.25 and 0.50, respectively. As the figure shows, the size of the robots' table membership depends on the swarm's size—that is, with a 1,000-robot swarm, each robot knows the membership of about

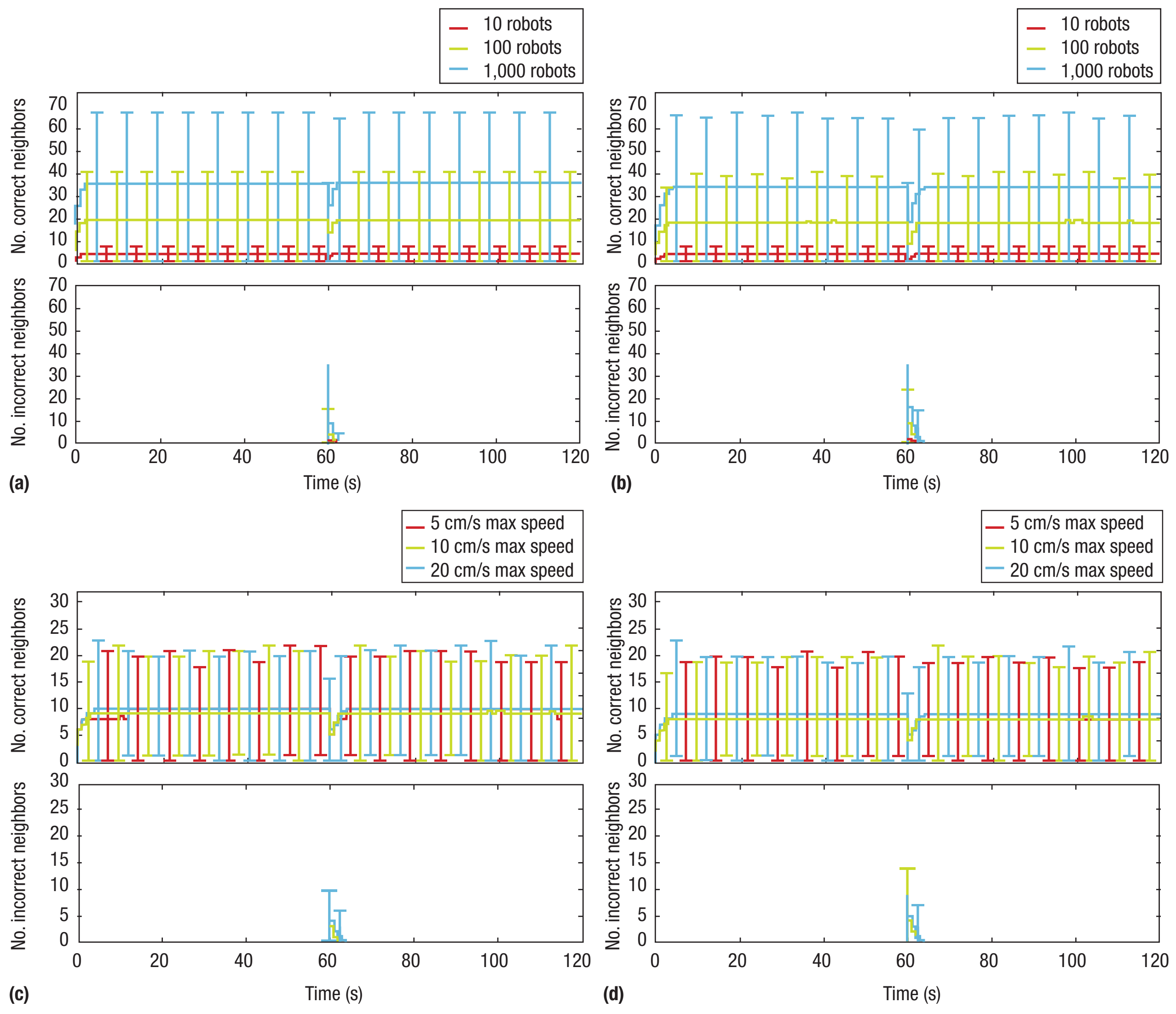


FIGURE 5. Assessment of the `swarm` construct. The plots illustrate the results from a series of experiments with varying robot topology, density, and packet drop rate: (a) static scale-free topology, 0.25 packet drop; (b) static scale-free topology, 0.50 packet drop; (c) dynamic topology, 0.25 packet drop, medium density; and (d) dynamic topology, 0.50 packet drop, medium density.

35 robots (median). When the robots switched from swarm 1 to swarm 2, the detection of the change was fast: with 1,000 robots, it took less than 5 seconds for the entire swarm to have no wrong entries in the membership tables.

Effect of motion

In the second set of experiments, we investigated how motion impacts the performance of swarm management. We initially distributed 100 robots uniformly in an environment that included obstacles (columns). The robots performed a simple diffusion algorithm. We then analyzed the effect on performance of two parameters: robot density and motion speed. We considered three levels for the density *D*: tight, medium, and sparse. We set the motion speed *M* maximum to 5 cm/s, 10 cm/s, and 20 cm/s. Each parameter choice (*D*, *M*, *P*) was tested in 30 runs for a total of 810 experiments.

Figures 5c and 5d report the results of (medium, 10 cm/s, 0.25) and (medium, 10 cm/s, 0.50), respectively. We can see that the swarm's density affects the size of the membership table more than the maximum speed: with a tight density, the median size was 11, 12, and 13 for 5 cm/s, 10 cm/s, and 20 cm/s, respectively. The same happened for the time it took the robots to update their membership tables, which was under 5 seconds for all the experiments.

In the experiments with the real robots, we replicated the same setup as the simulation and used Khepera IV LEDs to report changes in the swarm table. The aim of this experiment was to verify the correctness of the swarm table. The experiment was repeated 10 times using the Khepera IV robots, which reported correct swarm table results in all the experiments.

The use cases discussed here indicate that swarm membership is likely to change slowly—on the order of tens of seconds or more. Nevertheless, our results confirm that the `swarm` construct can be used in a variety of situations, involving large swarms with topologies that constantly change and with heavy message loss.

Through the `swarm` construct, we envision a world in which developers can specify the behavior of heterogeneous swarms of robots and package this behavior in an application that can be installed on multiple robotic systems. There will certainly be a market for these robotic apps, as manufacturers will be able to supply customers with robots and other devices that can be customized for specific activities. The `swarm` language construct is an important step toward the realization of this vision because it provides a simple mechanism for overcoming the global-to-local granularity choice.

Future work in this area includes the creation of an algebra that captures the main features of this concept and allows developers to make sound predictions about the behavior of a swarm before execution. C

ABOUT THE AUTHORS

CARLO PINCIROLI is an assistant professor at Worcester Polytechnic Institute. His research interests include software engineering and swarm robotics, and he is the author of the Buzz programming language and ARGoS (Autonomous Robots Go Swarming), a multirobot simulator widely used in swarm robotics research. Pinciroli received a PhD in applied sciences from Université Libre de Bruxelles. He is a member of IEEE, the IEEE Robotics and Automation Society, and ACM. Contact him at cpinciroli@wpi.edu.

GIOVANNI BELTRAME is an associate professor at École Polytechnique de Montréal. His research interests include the modeling and design of embedded systems, artificial intelligence, and robotics. Beltrame received a PhD in computer engineering from Politecnico di Milano. He is a member of IEEE, the IEEE Circuits and Systems Society, and ACM. Contact him at giovanni.beltrame@polymtl.ca.

REFERENCES

1. C. Pinciroli and G. Beltrame, "Buzz: An Extensible Programming Language for Heterogeneous Swarm Robotics," *Proc. IEEE/RSJ Int'l Conf. Intelligent Robots and Systems* (IROS 2016), 2016, pp. 3794–3800.
2. M. Brambilla et al., "Swarm Robotics: A Review from the Swarm Engineering Perspective," *Swarm Intelligence*, vol. 7, no. 1, 2013; link.springer.com/article/10.1007/s11721-012-0075-2.
3. M. Quigley et al., "ROS: An Open Source Robot Operating System," *Proc. ICRA Workshop on Open Source Software*, vol. 3, no. 2, 2009; www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.
4. L. Mottola and G. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art," *ACM Computing Surveys*, vol. 43, no. 3, 2011, article no. 19.
5. J. Beal et al., "Organizing the Aggregate: Languages for Spatial Computing," preprint, Cornell Univ. Library, 2012; arxiv.org/abs/1202.5509.
6. J. Bachrach, J. Beal, and J. McLurkin, "Composable Continuous-Space Programs for Robotic Swarms," *Neural Computing and Applications*, vol. 19, no. 6, 2010, pp. 825–847.
7. D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical Aggregate Programming," *Proc. ACM 30th Ann. Symp. Applied Computing* (SIGAPP 15), 2015, pp. 1846–1853.
8. M.P. Ashley-Rollman et al., "A Language for Large Ensembles of Independently Executing Nodes," *Logic Programming*, P.M. Hill and D.S. Warren, eds., LNCS 5649, Springer, 2009, pp. 265–280.
9. C. Pinciroli, A. Lee-Brown, and G. Beltrame, "A Tuple Space for Data Sharing in Robot Swarms," *Proc. 9th EAI Int'l Conf. Bio-inspired Information and Comm. Technologies* (BICT 2015), 2015, pp. 287–294.
10. A. Martinoli, K. Easton, and W. Agassounon, "Modeling Swarm Robotic Systems: A Case Study in Collaborative Distributed Manipulation," *Int'l J. Robotics Research*, vol. 23, no. 4, 2004, pp. 415–436.
11. K. Dantu et al., "Programming Micro-Aerial Vehicle Swarms with Karma," *Proc. 9th ACM Conf. Embedded Networked Sensor Systems* (SenSys 11), 2011, pp. 121–134.
12. L. Mottola et al., "Team-Level Programming of Drone Sensor Networks," *Proc. 12th ACM Conf. Embedded Network Sensor Systems* (SenSys 14), 2014, pp. 177–190.