SwarmTalk – Towards Benchmark Software Suites for Swarm Robotics Platforms

Yihan Zhang
Northwestern University
Evanston, IL, United States
yihan.zhang@u.northwestern.edu

Lyon Zhang
Northwestern University
Evanston, IL, United States
lyonzhang2021@u.northwestern.edu

Hanlin Wang
Northwestern University
Evanston, IL, United States
h.w@u.northwestern.edu

Fabián E. Bustamante
Northwestern University
Evanston, IL, United States
fabianb@cs.northwestern.edu

Michael Rubenstein
Northwestern University
Evanston, IL, United States
rubenstein@northwestern.edu

ABSTRACT

With nearly every new swarm robotic platform built, the designers develop its software stack, from low-level drivers to high-level algorithmic implementations. And while the different software stacks frequently share components, especially in robot-to-robot communication, these common components are also developed from scratch time and again. We present SwarmTalk, a new communication library that can be quickly ported to new and existing swarm hardware. SwarmTalk adopts a publish-subscribe communication model that satisfies the severe hardware constraints found in many swarms, and provides an easy-to-use programming interface. We port our SwarmTalk prototype to two hardware swarm platforms and two simulator-based platforms, and implement commonly-used swarm algorithms on these four platforms. We present the design and implementation of SwarmTalk, discuss some of the system challenges in implementation and cross-platform porting, and report on our initial experiences as a common communication abstraction for a community benchmarking suite.

KEYWORDS

swarm robotics; benchmarks; communication

ACM Reference Format:
Yihan Zhang, Lyon Zhang, Hanlin Wang, Fabián E. Bustamante, and Michael Rubenstein. 2020. SwarmTalk – Towards Benchmark Software Suites for Swarm Robotics Platforms. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, May 9–13, 2020,* IFAAMAS, 9 pages.

1 INTRODUCTION

Research in the field of swarm robotics seeks to coordinate the actions of a group of simple, physical robots. Visions of future swarm robots imagine them solving tasks with more reliability, adaptability, and scalability compared to traditional monolithic robots. Swarms are expected to be tolerant to individual failures, reconfigurable for easy adaptation to changing tasks, and scalable to match the expanding needs of a wide range of tasks. Some of the exciting applications for these systems include warehouse management, space exploration, search and rescue, self-driving vehicle navigation, crop pollination, and precision farming.

While research excitement with swarms goes as far back as the 1980s, recent advances in computing, communication, manufacturing and assembly are enabling researchers to realize physical swarms much larger than before. The last few years we have seen a growing number of swarm platforms being built with hundreds and even thousands of robots [11, 24, 30, 33, 36].

Each of these platforms, with unique hardware designs and limitations, currently requires the development of its own software stack, from low-level drivers, such as network I/O buffering or pulse-width modulation (PWM) for differential motors, to high-level algorithmic implementations. Once built, these platforms are commonly evaluated using a mostly ad-hoc set of algorithms and home-grown implementations.

As the field matures, the value of a common benchmarking suite to drive performance-based designs and platform comparison will become increasingly clear. Widely used for evaluating computer systems [39], in robotics benchmarking has so far focused on single robot performance [1], specific types of swarm robotics tasks [6], or on simulated robot systems [42, 44]. No current benchmarking suite addresses the hardware challenges of various swarm platforms in terms of computational and communication constraints. We argue that a common benchmark suite that includes a community agreed-upon set of benchmarks and is easily portable to any swarm platform, will help propel the field forward by enabling side-by-side comparisons of alternative solutions, both in terms of hardware designs and algorithm implementations, or between subsequent iterations of a system under evaluation.

Key to building a benchmarking system is identifying a portable and easy-to-use programming model. There have been efforts regarding Domain Specific Languages for programming and managing swarm robots [2, 31, 45] as well as OS abstractions for miniature robots [43]. Similar discussions have also happened in the Wireless Sensor Networks community [25]. However, most of these programming models rely on communication primitives provided by either the OS or the firmware. Most swarm platforms trade packet and memory size for cost and scalability, for instance by severely limiting the maximum transmission unit for communication. This complicates the porting of programming abstractions and algorithms, potentially requiring the use of data compression or packet

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 9–13, 2020, Auckland, New Zealand.

fragmentation, a far from trivial task. To the best of our knowledge, there is currently no protocol that fits in such small packets given the communication scenario, which will be discussed in Sec. 2.1.

As swarm behaviors are dominated by communication and sensing interaction between robots, we believe that a first step towards creating this benchmarking suite is to abstract away the communication layer of swarm platforms into a shared communication library with a common, agreed-upon interface to low-level drivers.

In this paper, we present SwarmTalk, a minimalistic communication library for extremely hardware-constrained swarm robotics platforms. SwarmTalk adopts a publish-subscribe communication model [10] and supports the transmission of multi-packet messages. Under the publish-subscribe communication model, robots wishing to send a message register first as publishers on a virtual channel to which other robots, interested in receiving the message, can register as subscribers. SwarmTalk is designed for portability. For each new hardware platform being built, the platform developers will only need to write the minimal amount of driver interface code once and can use the provided user interface for all future communication programming. This library can also serve as the communication component for the programming abstractions mentioned above. The system is designed to have a minimal footprint, with a user configurable packet header that is default to 3 bytes. We have verified our design through an open source implementation of the library, as well as the driver interface implementations for two physical robot platforms - Kilobot [36] and Coachbot V2.0 [46] - and two virtual platforms in ARGoS simulator [32] - Kilobot and Footbot [3]. All four robot environments have been tested with three canonical swarm behaviors to examine the performance in both static and mobile environment, as well as in time critical operations. With only interfacing the system-unique LED and motor actuation to a set of consistent function calls, the user code for implementing the behaviors is identical across all four platforms. This shows promising support for an open benchmark suite.

We envision a new phase for research in swarm robotics platforms in which a research group will develop a platform, link the low level drivers to the communication library, and automatically inherit all programs above the library level, such as benchmark suites or higher abstractions. The configurable library can provide a consistent base line to separate the algorithms and the hardware into two components, so as to compare their performance independently. Within a research community, benchmarks are a statement of the discipline's research goals, and they emerge through a synergistic process of both technical knowledge and community buy-in [39]. We expect that, much as high-level languages did in the 60-70s, a higher-level communication model may allow researchers to focus on the problem rather than the implementation details, and thus be able to tackle more challenging problems.

2 SWARMTALK DESIGN

Given the event-driven, many-to-many communication approach used in most applications of swarm robotics, SwarmTalk follows a publish-subscribe communication model [4, 9]. An alternative decoupled model of communication we considered is Linda-like distributed tuples [12], which offers a similar model for publishers and subscribers but has the added complexity of requiring support for a distributed tuple space. Publish-subscribe has been adopted by a wide-range of applications, from news aggregators (e.g., Twitter) to real-time systems (e.g., XMPP) and sensor networks [19]. Under this model, communicating nodes are referentially decoupled, that is, nodes are not individually identified. Instead, nodes declare their interest in particular data by subscribing to relevant topics or channels on which other nodes can publish. Furthermore, messages can be sent on different channels completely independently from each other. For instance, nodes can regularly publish the current value of their clock in one channel while only occasionally publishing externally detected events in another channel.

2.1 Design principles

The design of SwarmTalk is driven by realistic hardware requirements and assumptions, leveraging features unique to swarm robotics. The set of high-level design principles are described as follow.

a) Minimal resource requirement: The design aspect of swarm robotic hardware platforms usually emphasizes the cost and size of individual robot. The building cost of individual robots, both in terms of time and money, has limited the overall size of swarms to the range of a few tens or, at the very most, a few hundred robots. Only in recent years have new, low-cost miniature robot platforms been developed that make affordable experimentation with swarm sizes of up to a thousand robots in research labs possible. Coming with such low cost, the hardware capabilities are very limited, both in terms of communication and memory. In order to account for even the least capable robots, such as the Kilobot with 9 byte packets as the maximum transmission unit (MTU) and a 2048 KB SRAM, the library is required to use absolutely minimal resources. The packet headers should contain just enough information for the virtual channel and the packet fragmentation. At the same time, it should be as short as possible so that the already small MTU sacrifices as little of the payload as possible. The network buffer should be bounded by user configuration and should allocate all resources statically at compile time as described as a common technique for memory deficient sensor network nodes [20]. It is reasonable to assume that the cost for digital components will decrease and impose fewer resource constraints over time, but we envision smaller, cheaper and more scalable platforms to be built accordingly. This dichotomy still necessitates minimal resource requirement.

b) Dynamic Network Topology: Swarm robots are moving entities often with a very limited communication range. Using only a broadcasting model for the underlying network layer, the set of neighbors a robot can reach is constantly changing. The issue with such loosely connected topology is that, without any centralized broker similar to a DHCP server, it is at worst unfeasible and at best inefficient for individuals to distributively come up with globally unique IDs. This imposes challenges regarding bi-directional communication as well as packet routing. As a result, the library should only deliver an abstraction for multi-channel broadcasting. Summarily, the overhead should be small enough for compatibility with even the least capable platforms, while still allowing more complicated routing algorithms to be built upon it. To enable packet defragmentation at the receiving end, locally unique IDs are used instead. To account for the topology changes due to the robots' motion, each robot should be able to quickly switch to a unique

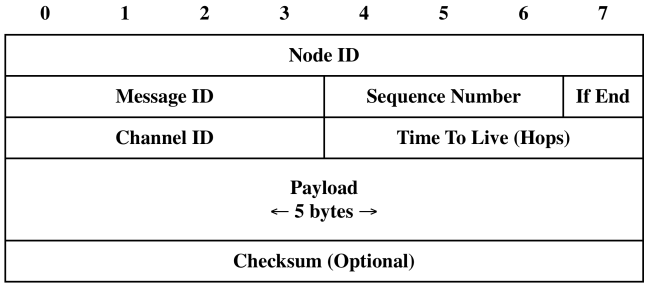

Figure 1: Default packet structure for SwarmTalk. Each row denotes a byte in the packet. With a 9 bytes MTU as illustrated, 3 bytes will be used for header and 1 byte will be used for checksum. The remaining 5 bytes will be used for fixed size payload.

ID if an ID collision happens within a one hop communication range. The size of the local ID space should be user configurable according to the ratio between the robot communication range and the robot radius. The ID space is recommended to be at least larger than the number of robots in the communication radius to avoid overwhelming ID collisions, but to be kept as small as possible to reduce the packet header overhead. The library should also provide multi-hop broadcasting with a decreasing Time To Live at each hop, so that each robot both serves as a consumer and a forwarder.

c) Situated Communication: Situated communication is often used in swarm robotics to provide sensing of the physical environment along with the corresponding message receptions [40]. For example, the distance between the sender and the receiver measured during transmission is provided by most platforms and is a necessary assumption for many algorithms. The distance is either achieved by measuring the signal strength at the receiving end or by using localization devices, such as GPS, to calculated the geographical difference. There are an arbitrary amount of sensing attributes to be measured by more capable hardware sensors, for example bearing (the direction of the incoming message) to enable algorithms that require direction sensitive communication [16]. To account for this variability, the library should allow platform developers to specify the data entries to be passed to the programmers. For multi-packet messages, the library should only provide the sensing for the most recently received packet.

d) Message delay overhead estimation: Any extra buffering of packets will introduce end-to-end delay overhead. Such underlying delay can be detrimental to time sensitive algorithms. For example, the firefly synchronization algorithm described in [47] requires an estimation of the delay between when a message is sent and when it is received. Thus the library should be able to use the clock provided by the platform to calculate the buffering delay both at the sending and the receiving end, so that users can have the realistic understanding of the end-to-end message delay. Since the sending delay and the receiving delay can only be captured according to sender's or receiver's local clock, the overall delay estimation can be inaccurate due to the clock skew. The design has to make an assumption that the captured delay would be relatively short, so that the clock skew between the sender and receiver won't impose too much of a difference on the estimation.

e) Portability: This communication library must be compatible with a wide variety of platform resources, from simplistic swarms

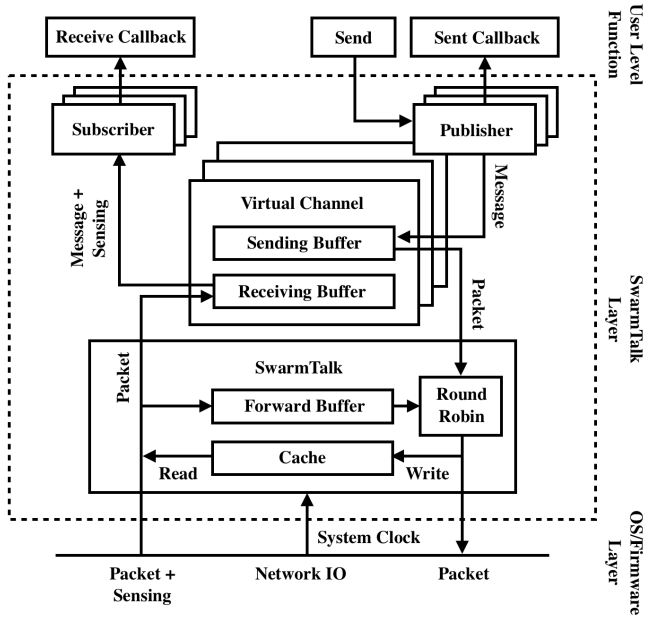

Figure 2: System components for SwarmTalk.

that use 9 byte packets to those using complex WiFi based communication. The design should thus make only basic assumptions about the platform drivers to make porting the library easy. The assumption including the following functionalities - interface for packet reception, interface for packet sending, and function for getting system clock. Once ported to new platforms, the library should allow existing communication programs to be run without much effort.

f) Ease of use: The communication interface for most platforms can be classified into two types:
(1) With a fixed size buffer for storing sending messages and a fixed size buffer for storing receiving messages. The user will have to update or pull the buffers regularly [3, 46].
(2) With event driven functions to provide a handler for fixed size receiving or sending buffer. [36]
Without using the library, users will have to carefully compress all the data to be sent, sometimes on the level of changing bits, to fit these platform specific constraints. This greatly raises the difficulty of porting any algorithm implementation to platforms designed with different communication interfaces or MTU. The library should provide a consistent communication interface and user experience across platforms.

2.2 System Components

With the described high level principles, we have developed the system with a default 3 byte header per packet. As shown in figure 1, the fields for the header include - `Node Id`, `Message Id`, `Sequence Number`, `If End`, `Channel Id`, and `Time To Live`. The bit length of each field is empirically set for the current version and can be configured by the user at compile time if needed. Extra bits will be padded if the configured header is not long enough for the payload to be byte aligned. The `Node Id` denotes the locally unique robot id and the `Message Id` denotes the randomly generated id for each new message. Both help serve the purpose of

identifying which packets should be composed back into a message. The `Sequence Number` and the `If End` indicates the sequence of packets assembling and if the current packet is the end of the sequence. The `Sequence Number` field also limits the maximum length of a message. For example, the default header only has 3 bits for the `Sequence Number`, meaning a maximum of 8 packets are allowed for each message. This is still much larger than the original maximum message size. Taking Kilobot for example, the payload of each packet is 6 bytes (9 bytes packet - 3 bytes header) which allows a maximum message length of 48 bytes (6 bytes per packet * 8 packets). If, in practice, other fields can be reduced to allow more bits for the `Sequence Number` with the same header size, each extra bit will double the maximum message length. The `Channel Id` specifies the virtual channel that the message is being sent on. The `Time To Live` is set to be the number of hops the packet should be forwarded. One byte of checksum is optional for a simple data integrity check.

The main components of the communication model include `SwarmTalk`, `Channels`, `Publishers` and `Subscribers`. As displayed in figure 2, one `SwarmTalk` object is responsible for all data transferring between the lower level network IO. It keeps a list of `Channel` objects with different `Channel IDs`, a `Forward Buffer` and a `Cache`. Each `Channel` object holds a list of `Subscriber` objects and a list of `Publisher` objects. Each `Channel` also manages its own `Sending Buffer` and `Receiving Buffer`. All memory is allocated at compile time and can be configured by the user. The user is required to initialize the `Subscribers` and `Publishers` with callback functions at run time for event driven programming. The `Subscribers` are designed with a default distance filter to ignore the messages received from a source too far away.

On message sending, the user can send messages of various length through the `Publisher`, which will check the availability of the `Channel Sending Buffer`. Only when the `Sending Buffer` is empty, will the `Channel` fragment the message into packets and store them into the `Sending Buffer`. The `SwarmTalk` object will use a round-robin method to pull packets from the `Forward Buffer` and the `Channels` when the network IO is ready to send. Each out-sending packet header will be stored into a `Cache`. When all packets in the `Sending Buffer` have been successfully processed, the corresponding `Sent Callback` function will be invoked.

On packet receiving, the `SwarmTalk` will first check the packet header against the `Cache` to make sure it did not recently send the packet, to prevent broadcast storms in multi-hop broadcasting [27]. If the packet has a `Time To Live` larger than 0, it will be copied to the `Forward Buffer` with decremented `Time To Live` for future forwarding. The `Channel` with the corresponding `Channel ID` will also obtain a copy of the packet and go through the `Receiving Buffer` to check for potential message defragmentation. Once messages are formed from the packets, the `Receive Callback` function will be invoked.

We have developed the `Channel` with an option to keep a timestamp to estimate the end to end delay caused by buffering on each packet. This feature trades off extra bytes from the payload, but is necessary for time critical operations.

2.3 User Interface

The current SwarmTalk assumes a C++ programming interface with a `setup` function called at robot start and a `loop` function called periodically, which are very common embedded system programming practices. This section will provide a walk through of how to build a simple program for robots to broadcast a tuple consisting of a randomly generated ID and the local clock. For this demonstration, `struct` is used to store the tuple because it can be easily serialized into unsigned char arrays.

Assume we have the following as local variables. Since both `random_id` and `clock` are 64 bits, or 8 bytes, there is no trivial way for the Kilobot to send such a tuple using a 9 bytes MTU without using the library.

```
typedef struct state {
   uint64_t random_id;
   uint64_t clock;
} state_t;

Channel *channel;
Publisher *publisher;
Subscriber *subscriber;
state_t my_state;
```

Now let's initialize these variables in the `setup` function. We are here assuming the channel ID is 0, packets are only transmitted within one hop, no timestamp will be added to the packets, and messages farther away than 100 (platform specific distance sensor reading) will be filtered.

```
void setup() {
  // Initialize the random_id with rand()
  my_state.random_id = rand();
  // Initialize the clock with the system clock
  my_state.clock = get_clock();
  // Initialize the channel pointer from the swarmtalk
  // First parameter 0 indicates the channel ID
  // Second parameter 0 indicates the Time to Live
  // Third parameter false means this channel does not put timestamp on packets
  channel = swarmtalk->new_channel(0, 0, false);
  // Initialize the publisher with the callback function sent
  publisher = channel->new_publisher(sent);
  // Initialize the subscriber with the callback function recv
  // First parameter 100 filters out messages that are more than 100 units away
  subscriber = channel->new_subscriber(100, recv);
  // Send the serialized message using the publisher
  publisher->send((unsigned char *) &my_state, sizeof(my_state));
}
```

To keep sending the most up-to-date tuple, we can define the `sent` callback to update the clock and send again.

```
void sent() {
  // update the clock
  my_state.clock = get_clock();
  // send the updated message again
  publisher->send((unsigned char *) &my_state, sizeof(my_state));
}
```

We can define the `recv` callback to parse the message and check if the values are correct by printing them out. `Data_t` is defined by the user to encompass all the sensing data for situated communication.

```
void recv(unsigned char * msg, int size, int ttl, Data_t * sensing) {
  // Deserialize the message
  state_t * recv_state = (state_t *) msg;
  // Print the ID field of the tuple
  std::cout << "ID: " << recv_state->random_id << std::endl;
  // Print the clock field of the tuple
  std::cout << "Clock: " << recv_state->clock << std::endl;
  // Print the measured distance provided by the lower level driver
  std::cout << "Distance from sender: " << meta->dist << std::endl;
}
```

3 IMPLEMENTATION

We have implemented the SwarmTalk system in C++ with a small set of benchmarking algorithms as an open source project. The

source code is public and hosted at: https://github.com/shzhangyihan/SwarmTalk.

There are currently four supported environments: Kilobot, Coachbot V2.0, simulated Kilobot in ARGoS simulation and simulated Footbot in ARGoS simulation (see figure 3). To integrate SwarmTalk to new platforms, it is as easy as implementing four function handles: (1) a function to pull out-sending packet from SwarmTalk when network IO available; (2) a function to notify SwarmTalk on packet reception; (3) a function for random number generation; (4) and a function for local clock.

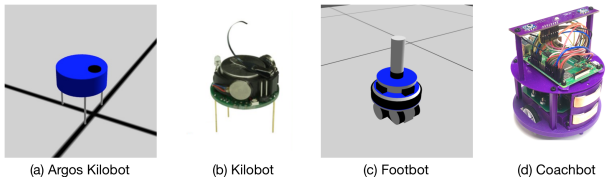

Figure 3: The robots of the four evaluated platforms.

3.1 Kilobot

Kilobot is the first platform to demonstrate experiments with 1024 robots performing self-assembly algorithms, and shows the possibility of creating affordable large scale robot platforms [37]. Each Kilobot costs $14 of parts and 5 minutes of assembling, significantly lower than any counterparts at the time [36].

The robot-to-robot communication is achieved by each robot broadcasting packets to neighbors (up to 6 robot radii) using infrared transmitters and photodiode receivers in a CSMA/CA manner. The bandwidth of such communication channel is often reduced due to the collisions when the density of robots within communication range increases. Such hardware characteristics lead to the communication stack with 9-byte long packets that, by default, attempts to send messages twice per second.

3.2 Coachbot V2.0

Coachbot V2.0 is a more powerful platform that uses a Raspberry Pi 3b+ with dual-band 802.11ac wireless LAN for each robot. To achieve robot-to-robot communication, layer 2 broadcasting (i.e. MAC address based broadcasting) is used to avoid having a centralized base router. The MTU for Ethernet frames, 1500 bytes by default, is significantly larger than that of its counterparts, so we limit it to have the same MTU size as the Kilobot for demonstration purposes with fair comparisons. Each robot has sensors to receive infrared signals from the ceiling mounted HTC vive lighthouse for localization. The position information of the sending robot is embedded in the data packets for the receiving robots to calculate the distance [46]. For the simplicity of explanation, the Coachbot V2.0 will be referred as Coachbot for the rest of the paper.

3.3 ARGoS

ARGoS is a realistic physics-based multi-robot simulator that splits robots into their component actuators and sensors. Despite having different designs, many robots have similar actuators and sensors, such as motors powering wheels, LED's, and infrared sensors. By taking these common components and assembling them in different ways, ARGoS can simulate a wide variety of real robots. [32]

The two simulated robots we are considering here are the Kilobot and the Footbot. The simulated Kilobots provide an identical programming interface as well as very similar experimental performances compared to the physical ones. The Footbot, or the MarXbot, is another swarm platform with more advanced battery management and an omnidirectional camera [35]. The simulated Footbot communicates with its neighbors using a range-and-bearing communication device proposed in [3], that uses infrared with a default packet size of 10 bytes as MTU. The two platforms will be refered as Argos Kilobot and Footbot for the rest of the paper.

4 EVALUATION

We identify a starting set of fundamental algorithms or primitives commonly used in the field, and add the implementations to a benchmarking suite prototype. We implement three algorithms - *hop count*, *firefly synchronization* and *relative motion* - each with identical user program across the platforms. We select these algorithms as our initial set as they are representative of the type of primitives used as building blocks for more elaborate solutions like shape formation. Each of the algorithms is implemented using SwarmTalk to demonstrate the different functional aspects of the library. To ensure fair comparison among the platforms, all of them are configured to the same packet format - 9 bytes MTU with 3 bytes header and 1 byte of checksum, leaving 5 bytes for the payload. The result is evaluated in terms of (1) if the task can be accomplished with straightforward user code and (2) if the performance is consistent for the four platforms, demonstrating the potential of SwarmTalk serving as the building block for benchmark suites. All experiments are performed successfully 10 times each without failing. Videos of the recorded experiments can be found at: https://vimeo.com/373516798

4.1 Hop Count

The hop count algorithm is frequently used in swarm robotics to measure network distance (and therefore an approximation of physical distance). In this simple algorithm, all robots in the swarm measure the minimum number of re-transmissions (or hops) required for a message to reach it. This is implemented by having a single seed send out a message to its neighbors with a hop value equal to 0. All other robots keep track of the lowest hop value they receive and update the local hop count with the minimal value + 1. All robots keep sending its hop count to all the neighbors. This simple algorithm propagates messages from robot to robot and measures the minimum number of hops for each robot to the seed. This algorithmic primitive is used in a wide variety of behaviors, including allowing motion without network disconnection [37], trilateration based localization [26], group motion control [38], and group event detection [13].

Two sets of experiments are conducted to demonstrate the capability of multi-channel communication and the advantage of channel-based event-driven programming. All pictures are recorded using a ceiling camera for physical platforms, or through screen captures for simulated platforms, 10 seconds after the program starts running to capture the converged state. For the first set of experiments, 36 robots are placed to form a compact square shape for all four platforms, with the upper left robot as the only seed. Only

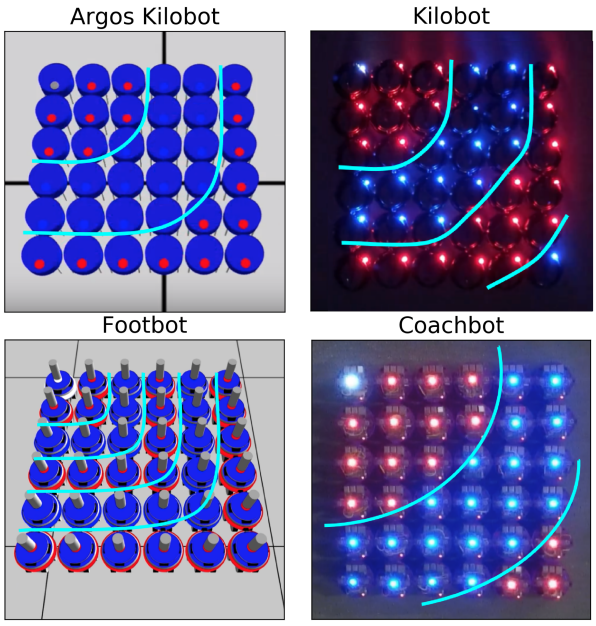

Figure 4: Gradients generated using the hop count algorithm, with a single seed robot placed in the upper left corner for each platform. The color displayed by each robot's LED represents the hop count, where red means the hop count is an even number and the blue means otherwise. The lines were manually added to the images and denote approximate single-hop boundaries. Although communication range and shape differs from robot to robot, a uniquely distinct gradient forms on all four platforms.

one channel is used in this experiment for broadcasting robots' hop count from the seed. The user level program can be implemented as following, where the `sent()` and the `recv()` are the set of functions registered as callbacks for the publisher and the subscriber of the only channel.

```
typedef struct custom_message {
    int hop;
} custom_message_t;
Channel * channel;
Publisher * publisher;
Subscriber * subscriber;
custom_message_t my_message;

void sent() {
  // after sending to channel 1 finished, start new send to channel 1
  publisher->send((unsigned char *) &my_message, sizeof(my_message));
}

void recv(unsigned char * msg, int size, int ttl, Data_t * sensing) {
  // update the hop and the LED
  custom_message_t * received_msg = (custom_message_t *) msg;
  if(received_msg->hop < my_message.hop) { my_message.hop = received_msg->hop; }
}

void loop() { }

void setup() {
  my_message.hop = 100;
  channel = swarmtalk->new_channel(0, 0, false);
  publisher = channel->new_publisher(sent);
  subscriber = channel->new_subscriber(250, recv);
  publisher->send((unsigned char *) &my_message, sizeof(my_message));
}
```

As shown in figure 4, clear gradients, as alternations between red and blue LEDs, are displayed for all platforms, showing the convergence of the hop count algorithm with one seed.

For the second set of experiments, robots are with the same placement, but with two seeds instead. Assume one seed at the

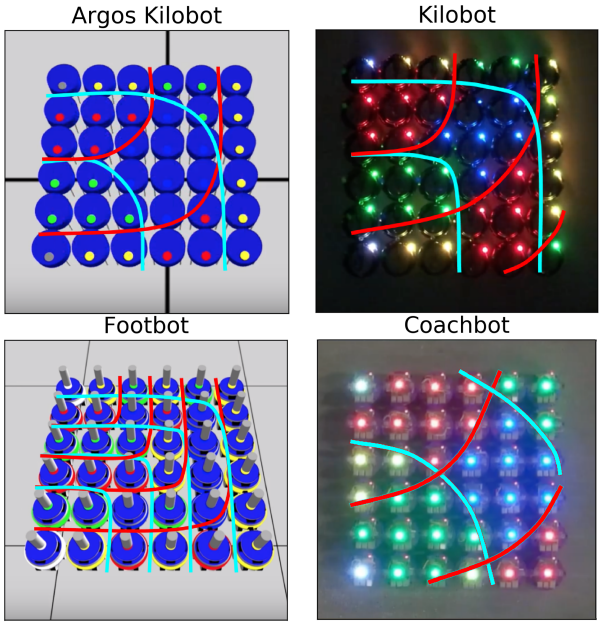

Figure 5: Two-seed gradients generated on all platforms using the hop count algorithm. The seeds are placed in the upper left and lower left corners of the array in all cases. The LED has four different colors to present four states - with red meaning the hop counts from seed_1 is odd and from seed_2 is even; yellow meaning the hop counts from both seed_1 and seed_2 are odd; blue meaning the hop counts from both seed_1 and seed_2 are even; green meaning the hop counts from seed_1 is even and from seed_2 is odd. The red lines denote the single-hop boundaries from seed_1 and the cyan lines denote the boundaries from seed_2.

upper left corner as the `seed_1` and the other seed at the lower left corner as the `seed_2`. The previous implementation can be easily modified to have two channels for the robots to communicate the hop count from different seeds separately. The `sent_1()` and the `recv_1()` functions are the callbacks for one of the channels to communicate the hop counts from `seed_1`, while the `sent_2()` and the `recv_2()` are for the other channel for hop counts from `seed_2`.

```
// after sending to channel 1 finished, start new send to channel 1
void sent_1() { }
// after sending to channel 2 finished, start new send to channel 2
void sent_2() { }

// update the hop and the LED from seed_1
void recv_1(...) { }
// update the hop and the LED from seed_2
void recv_2(...) { }

void loop() { }
void setup() { }
```

The result is shown in figure 5, with all robots accurately keep track of two distance values, showing the reliability and accuracy of dual SwarmTalk channels. The hop count convergence from `seed_1` and from `seed_2` are essentially parallel operations that are independent of each other. That means adding more parallel communication tasks is as simple as adding more channels with pairs of event-driven functions. Without using the library, the same programming task can be achieved by adding differentiators in the packet payload and using conditional logic at the receiving end to

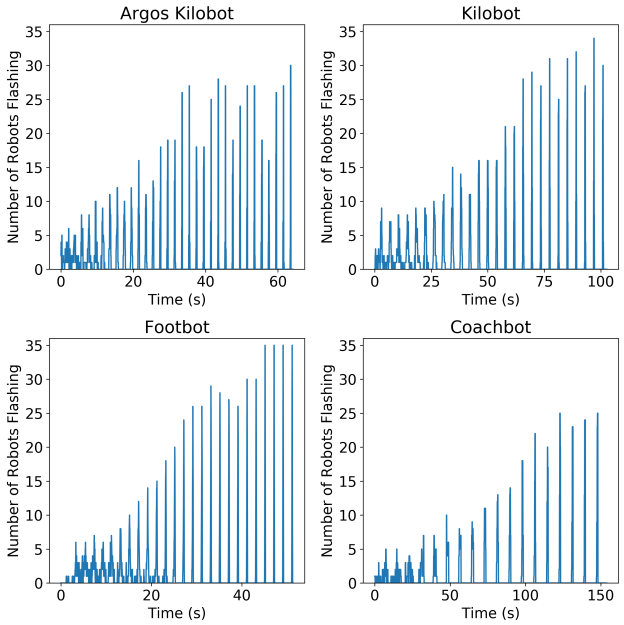

Figure 6: Time series graphs showing convergence patterns for the firefly algorithm across all four platforms. Here, the X axis is experiment time, and the Y axis is the number of robots "firing" during that time. Initially, robots start out of synchronization, but as the experiment progresses robots synchronize and fire all at nearly the same time.

identify the content. However, this method adds extra complexity for programmers, especially when the types of messages increase.

4.2 Firefly Synchronization

A wide range of swarm behaviors such as locomotion [38, 48], shared-channel communication [7], and distributed decision making [8], all rely on synchronization in time between robots throughout the swarm. One commonly used approach to create the time synchronization is using the synchronization of pulse-coupled biological oscillators [23, 29]. Here robots can synchronize their clocks by sending “synchronization” messages to their neighbors on “flashing”. These neighbors then update their period length, trying to match the next “flashing” of their neighbors. Since the firefly synchronization requires good estimation of the delay between when the “firing” takes place, i.e. the message sent, and when the neighbors “perceive” it, i.e. the message reception [47], it is a great example to examine the timestamp delay measurement method used by SwarmTalk. To manually increase the delay to make this test more challenging, each robot is sending a dummy message with 4 bytes using a channel with 4 bytes of timestamp. As each packet only has 1 byte remaining (5 bytes of payload - 4 bytes of timestamp), each “flashing” message will require 4 packets to be delivered.

The experiment is conducted with the same robot placement as the hop count one, with 36 robots forming a square and starting to flash randomly. Each time a robot flashes, it will send the 4-packet dummy message to all of its near neighbors. Upon message reception, neighboring robots will estimate their clocks by the time the message was actually sent, by subtracting the message buffering delay provided by SwarmTalk to the current clock, and adjust their

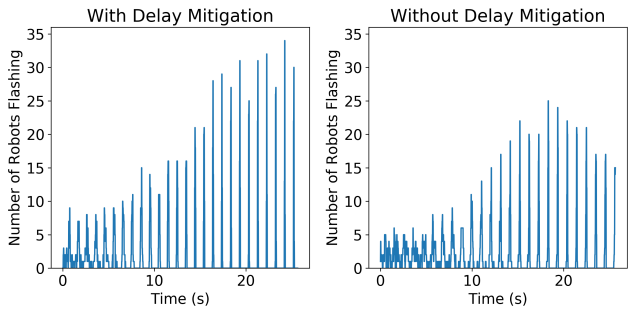

Figure 7: Comparison of convergence between Kilobots using the estimated buffering delay to mitigate message delay (left) and Kilobots not mitigating message delay (right). The Kilobots not not mitigating the delay never converge as well as the ones doing so.

period length accordingly. The video is recorded by a ceiling camera or using screen capture. The result is evaluated by counting the number of robots flashing at any given time, as shown in figure 6. The higher the number of robots flashing at the same time, or the y-axis value, the better the swarm is synchronized. Starting from a randomly flashing state, indicated by the noisy beginning of each plot, all four platforms successfully converged to a synchronized state as indicated by the clean vertical spikes at the end of the plot. Since clocks are skewed among robots, a few robots will still flash slightly out of sync. Therefore the curves realistically never reach 36, the total number of robots.

In order to determine the effectiveness of the estimated buffering delay, we also conduct a comparison experiment between the baseline implementation that uses the buffering delay to mitigate the message delay (same as the previous implementation) and the implementation that ignores the buffering delay. Both experiments use the same 36 physical Kilobots and send 4-packet messages when flashing, thus having the same amount of buffering delay. However, when the receive callback functions are invoked with the buffering delay as argument, the baseline implementation subtracts the delay from the current clock, while the implementation without delay mitigation just uses the unchanged current clock. From figure 7, the baseline implementation stays shorter at the randomly flashing state, where as the other one exhibits noisy curves until almost 10 seconds, and converges better, with higher y-axis values than that of the other one. From our observation of the video, without any delay mitigation, the message delay cause flashing “waves” that move across the robots and stop the robots from synchronizing, in comparison to the baseline implementation where the flashing are synchronized good enough to not see any wave.

4.3 Relative Motion

Individuals within a swarm are often required to move in position relative to their neighbors, for example in reconfiguration [48] or during shape formation [37]. This is often further complicated by the lack of global position sensing available to individuals. As a result, robots within the swarm frequently control their motion based on sensing relative information about their neighbors, such as heading and distance. This behavior is manifested in many different applications and approaches, from flocking, where aerial vehicles [17]

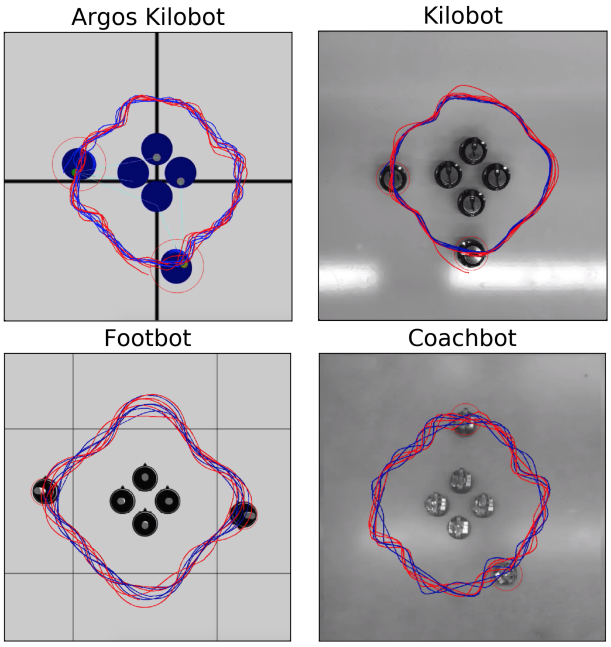

Figure 8: Red and blue lines mapping the paths of robots from all four platforms for the edge following algorithm. All images were taken after tracking robot movement for five complete revolutions.

or ground robots [18] maintain distance and heading to neighbors, to shape motion of individuals in shape formation [5, 37, 41].

The task for relative motion is defined as edge following. There are four `seed` robots placed in the middle as stationery anchors and two `follower` robots trying to go clockwise around the `seeds` at a fixed distance. Only distance sensing is provided during communication. Each `seed` can not talk to other `seeds`, and each `follower` can not talk to other `followers`. To avoid collision, each `seed` can only be followed by one `follower` while other `followers` trying to follow the same `seed` stop and wait. This edge following behavior can test the performance of SwarmTalk under a dynamic network topology as well as the ability to provide situated communication.

The implementation of this algorithm is achieved by creating two channels, one for the `followers` to send their states to the `seeds` and the other for the `seeds` to send the states to the `followers`. This implementation demonstrates the use case of having two unidirectional channels for two types of agents, with each only publishing to one of the channels and subscribing to the other. The experiment video is recorded using a ceiling camera or with screen capture for at least 5 complete revolutions. The video is then mapped with red and blue lines indicating the path of two `followers` as in figure 8. The resulting path graphs show successful edge following behavior, with clear similarities among the four platforms.

5 RELATED WORK

There have been works in the field of robotics and sensor network for system level abstractions trying to hide lower level details for easier and platform independent user experience. Some of the most prominent systems include ROS [34] and TinyOS [21]. ROS is the most commonly used robotics middleware and uses a topic based publish-subscribe model for communication. However, ROS is not suitable for fully distributed swarm platforms due to its centralized nature. ROS2 is a recent advancement to address the increasing need for the distributed multi-agent scenarios using the Real-Time Publish-Subscribe protocol (RTPS) by the OMG Data-Distribution Service as the new underlying communication architecture [28]. Even with a decreased hardware requirement, neither ROS2 nor its underlying RTPS protocol can fit onto hardware as simplistic as, for example, the Kilobot.

TinyOS is an event-driven operating system for embedded wireless devices. TinyOS uses nesC language to statically link all callbacks at compile time to achieve fully non-blocking operation [14]. However, TinyOS is rarely used in robotics platforms where actuation appears to be much more common than seen in the wireless sensor nodes. The default communication formats used by TinyOS are also too heavy to be fit onto the Kilobot. Other operating system abstractions include OpenSwarm [43], which provides a multithreading abstraction to allow more complex programs and to make better use of the computational resources.

6 CONCLUSIONS AND FUTURE WORK

We presented SwarmTalk, the first communication library for highly resource-constrained and mobile swarm robotics platforms. We have demonstrated the portability of SwarmTalk and its easy-to-use API. SwarmTalk-based implementations of different canonical algorithms perform almost identically across different platforms, making the case for SwarmTalk as the basis for a cross-platform benchmark suite.

While SwarmTalk has proven to be powerful for the problems we have tackled thus far, there are a number of open questions. In our current design, there are two types of channels - the general-purpose channel and the time channel with buffering delay estimation. A few commonly-used and purpose specific channels, such as debug or data harvesting channels, may prove useful and offer a good trade-off between generality and efficient implementation. Swarm-level operations, such as group re-program, or complicated network protocols, such as routing, should also be explored to see if they should be implemented as specialized channels or if they are orthogonal to the communication model.

It would be valuable to investigate more complicated algorithms for potential inclusion in the benchmark suite, such as leader election and generating coordinate system. Leader election allows a swarm of homogeneous decentralized robots to use a leader to make a centralized decision, coordinate actions [15], or seed a distributed coordinate system [41]. Coordinate systems, on the other hand, are frequently used in tasks that require precise global control, such as shape formation [37], [13], message routing [22], or providing robot-to-robot sensing [17]. Current packet and memory configurations have thus far been empirically determined; we plan to evaluate different settings to further relax the hardware constraint and improve performance.

REFERENCES

[1] Jacky Baltes. 2000. A benchmark suite for mobile robots. In *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*, Vol. 2. IEEE, 1101–1106.
[2] Jacob Beal and Jonathan Bachrach. 2006. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems* 21, 2 (2006), 10–19.

[3] Michael Bonani, Valentin Longchamp, Stéphane Magnenat, Philippe Rétornaz, Daniel Burnier, Gilles Roulet, Florian Vaussard, Hannes Bleuler, and Francesco Mondada. 2010. The marXbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 4187–4193.

[4] A. Carzaniga, M. Rutherford, and A. Wolf. 2001. Design and evaluation of a wide-area event notification service. *IEEE/ACM Transactions on Computer Systems* 3, 19 (2001).

[5] Jimming Cheng, Winston Cheng, and Radhika Nagpal. 2005. Robust and self-repairing formation control for swarms of mobile agents. In *AAAI*, Vol. 5.

[6] Micael S Couceiro, Patricia A Vargas, Rui P Rocha, and Nuno MF Ferreira. 2014. Benchmark of swarm robotics distributed techniques in a search task. *Robotics and Autonomous Systems* 62, 2 (2014), 200–213.

[7] Julius Degesys, Ian Rose, Ankit Patel, and Radhika Nagpal. 2007. DESYNC: self-organizing desynchronization and TDMA on wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*. ACM, 11–20.

[8] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)* 34, 1 (1987), 77–97.

[9] Greg Eisenhauer, Fabián E Bustamante, and Karsten Schwan. 2000. Event services for high performance computing. In *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 113–120.

[10] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *ACM computing surveys (CSUR)* 35, 2 (2003), 114–131.

[11] Nicholas Farrow, John Klingner, Dustin Reishus, and Nikolaus Correll. 2014. Miniature six-channel range and bearing system: algorithm, analysis and experimental validation. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 6180–6185.

[12] David Galernter and Nicholas Carriero. 1992. Coordination languages and their significance. *Commun. ACM* 35, 2 (1992), 96–108.

[13] Melvin Gauci, Radhika Nagpal, and Michael Rubenstein. 2018. Programmable self-disassembly for shape formation in large-scale robot collectives. In *Distributed Autonomous Robotic Systems*. Springer, 573–586.

[14] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. 2014. The nesC language: A holistic approach to networked embedded systems. *Acm Sigplan Notices* 49, 4 (2014), 41–51.

[15] Kyle Gilpin, Kent Koyanagi, and Daniela Rus. 2011. Making self-disassembling objects with multiple components in the robot pebbles system. In *2011 IEEE International Conference on Robotics and Automation*. IEEE, 3614–3621.

[16] Roderich Groß, Stéphane Magnenat, and Francesco Mondada. 2009. Segregation in swarms of mobile robots based on the brazil nut effect. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 4349–4356.

[17] Sabine Hauert, Severin Leven, Maja Varga, Fabio Ruini, Angelo Cangelosi, Jean-Christophe Zufferey, and Dario Floreano. 2011. Reynolds flocking in reality with fixed-wing robots: communication range vs. maximum turning rate. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 5015–5020.

[18] Suranga Hettiarachchi and William M Spears. 2009. Distributed adaptive swarm for obstacle avoidance. *International Journal of Intelligent Computing and Cybernetics* 2, 4 (2009), 644–671.

[19] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. 2003. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking* 1 (2003).

[20] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A Brewer, and David E Culler. 2004. The Emergence of Networking Abstractions and Techniques in TinyOS.. In *NSDI*, Vol. 4. 1–1.

[21] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.

[22] Martin Mauve, Jorg Widmer, and Hannes Hartenstein. 2001. A survey on position-based routing in mobile ad hoc networks. *IEEE network* 15, 6 (2001), 30–39.

[23] Renato E Mirollo and Steven H Strogatz. 1990. Synchronization of pulse-coupled biological oscillators. *SIAM J. Appl. Math.* 50, 6 (1990), 1645–1662.

[24] Francesco Mondada, Michael Bonani, Xavier Raemy, James Pugh, Christopher Cianci, Adam Klaptocz, Stephane Magnenat, Jean-Christophe Zufferey, Dario Floreano, and Alcherio Martinoli. 2009. The e-puck, a robot designed for education in engineering. In *Proceedings of the conference on autonomous robot systems and competitions*, Vol. 1. IPCB: Instituto Politécnico de Castelo Branco, 59–65.

[25] Luca Mottola and Gian Pietro Picco. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)* 43, 3 (2011), 19.

[26] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. 2003. Organizing a global coordinate system from local information on an ad hoc sensor network. In *Information processing in sensor networks*. Springer, 333–348.

[27] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. 1999. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. ACM, 151–162.

[28] Gerardo Pardo-Castellote. 2003. Omg data-distribution service: Architectural overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 200–206.

[29] Fernando Perez-Diaz, Stefan M Trenkwalder, Rüdiger Zillmer, and Roderich Groß. 2018. Emergence and inhibition of synchronization in robot swarms. In *Distributed Autonomous Robotic Systems*. Springer, 475–486.

[30] Daniel Pickem, Paul Glotfelter, Li Wang, Mark Mote, Aaron Ames, Eric Feron, and Magnus Egerstedt. 2017. The robotarium: A remotely accessible swarm robotics research testbed. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 1699–1706.

[31] Carlo Pinciroli and Giovanni Beltrame. 2016. Buzz: An extensible programming language for heterogeneous swarm robotics. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3794–3800.

[32] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, et al. 2011. ARGoS: a modular, multi-engine simulator for heterogeneous swarm robotics. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 5027–5034.

[33] James A Preiss, Wolfgang Honig, Gaurav S Sukhatme, and Nora Ayanian. 2017. Crazyswarm: A large nano-quadcopter swarm. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 3299–3304.

[34] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[35] James F Roberts, Timothy S Stirling, Jean-Christophe Zufferey, and Dario Floreano. 2009. 2.5 D infrared range and bearing system for collective robotics. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 3659–3664.

[36] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. 2012. Kilobot: A low cost scalable robot system for collective behaviors. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 3293–3298.

[37] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. 2014. Programmable self-assembly in a thousand-robot swarm. *Science* 345, 6198 (2014), 795–799.

[38] Wei-Min Shen, Behnam Salemi, and Peter Will. 2002. Hormone-inspired adaptive communication and distributed control for CONRO self-reconfigurable robots. *IEEE transactions on Robotics and Automation* 18, 5 (2002), 700–712.

[39] Susan Elliott Sim, Steve Easterbrook, and Richard C Holt. 2003. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 74–83.

[40] Kasper Støy. 2001. Using Situated Communication in Distributed Autonomous Mobile Robotics.. In *SCAI*, Vol. 1. Citeseer, 44–52.

[41] Kasper Stoy and Radhika Nagpal. [n. d.]. Self-repair through scale independent self-reconfiguration. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 2. IEEE, 2062–2067.

[42] Jose M Such, Juan M Alberola, Luis Mulet, Agustin Espinosa, Ana Garcia-Fornes, and Vicent Botti. 2007. Large-scale multiagent platform benchmarks. *LADS* (2007), 192–204.

[43] Stefan M Trenkwalder, Yuri Kaszubowski Lopes, Andreas Kolling, Anders Lyhne Christensen, Radu Prodan, and Roderich Groß. 2016. OpenSwarm: an event-driven embedded operating system for miniature robots. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 4483–4490.

[44] Richard Vaughan. 2008. Massively multi-robot simulation in stage. *Swarm intelligence* 2, 2-4 (2008), 189–208.

[45] Mirko Viroli, Jacob Beal, Ferruccio Damiani, Giorgio Audrito, Roberto Casadei, and Danilo Pianini. 2018. From field-based coordination to aggregate computing. In *International Conference on Coordination Languages and Models*. Springer, 252–279.

[46] H. Wang and M. Rubenstein. 2020. Shape Formation in Homogeneous Swarms Using Local Task Swapping. *IEEE Transactions on Robotics* (2020), 1–16. https://doi.org/10.1109/TRO.2020.2967656

[47] Geoffrey Werner-Allen, Geetika Tewari, Ankit Patel, Matt Welsh, and Radhika Nagpal. 2005. Firefly-inspired sensor network synchronicity with realistic radio effects. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*. ACM, 142–153.

[48] Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S Chirikjian. 2007. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics & Automation Magazine* 14, 1 (2007), 43–52.