2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 25-29, 2020, Las Vegas, NV, USA (Virtual)

# An Actor-based Programming Framework for Swarm Robotic Systems

Wei Yi[1,2,3],Bin Di[1,2],Ruihao Li[1,2],Huadong Dai[1,3,*],Xiaodong Yi[1,3],Yanzhen Wang[1,2,3],Xuejun Yang[1,3,*]

***Abstract*— Programming cooperative tasks for autonomous swarm robotic systems has always been challenging. In this paper, we introduce a concept 'Actor', as a virtualization for robot platforms. Every robot platform in the swarm robotic system carries out the task and interacts with others as an Actor. We designed an Actor-based framework for the management of autonomous swarm robotic systems including modules and interfaces for the Actor, the collective Actor, and task management. The Actor-based framework enables task developers to explicitly model cooperative tasks without intricacies about the detailed robotic algorithms or the specific robot brands, and eases the burden on robotic algorithm developers by providing common functionalities. The proposed framework is implemented in C++ and validated quantitatively and qualitatively with a swarm of thirty drones by simulations and a swarm of ten drones by in-field tests.**

## I. INTRODUCTION

Swarm robotic systems, where each individual equipped with a variety of sensors and executors, can cooperate delicately to complete complicated tasks. Many applications, such as target searching [1], disaster rescuing [2], and geographic mapping [3], benefit from the use of swarm robotic systems. In comparison with a single robot system, swarm robotic systems have advantages in collective intelligence, fault-tolerance capability, and high efficiency. Sensors with limited precision or range can be compensated by data fusion with swarm robotic systems. The task of individuals can be continued by other redundancy nodes in the swarm robotic system in the case of failure upon interference or hostile interruption. The efficiency of swarm robotic systems with good scalability can be improved by increasing the size of the system.

The swarm robotic system is a typical example of the 'System of systems'. Developing software for such a complex system is inherently complicated. To unlock the advanced features of swarm robotic systems in applications, programming is challenging work. Unfortunately, available tools for manipulating a swarm robotic system are rare. Projects that use swarm robotic systems were solved case by case, lacking the flexibility for application users to modify the task. In fact, task developers for swarm robotic systems only concern about collective behaviors of the system and generally give orders to the whole system instead of specific robot platforms. Therefore intricacies of underlying robotic algorithms and inter-robot communications should be hidden as much as possible for task developers.

In this paper, we introduce the concept 'Actor', which is the virtualization of robot platforms. The Actor encapsulates robotic hardware resources and a group of algorithm plugins that make use of these resources to manipulate robot behaviors. A robot platform carries out the task and interacts with others as an Actor. The Actor helps in decoupling the task from platforms in the swarm robotic system. Our research exploits features of swarm robotic systems by developing an Actor-based framework. The proposed framework aims to enable task developers to explicitly model the task and manipulate the system without intricacies about the detailed robotic algorithms or the specific robot brands, and ease the burden on algorithm developers by providing functionalities such as autonomous member joining/leaving detection, reorganization, conflict resolving, etc.

Our contribution can be summarized as below:

- We introduce a concept of Actor, to represent the high-level virtualization for robot platforms. Each designed Actor maintains a data structure, is bounded to different plug-in groups, and is a basic unit for the management of collective behaviors.
- We propose a mechanism for collective Actor management. Robot platforms in a swarm robotic system are organized effectively. Primitives for cooperative tasks such as Actor-level synchronization are derived naturally.
- We propose a domain-specific language (DSL) for composing Actor-based tasks. Task developers can be relieved from unnecessary details of manipulating individual robots, and focus on complex swarm robotic task coordination strategies.
- We implement the proposed framework in C++, and validate it quantitatively and qualitatively by both simulations and in-field tests.

## II. RELATED WORK

Robots nowadays have been introduced in many industrial and service applications. Designing a flexible programming model to develop robot algorithms and applications would greatly boost the development of robotics. Robot programming methods have attracted great interest in the last decade. A wide range of tools and libraries were developed, such as Miro [4], OROCOS [5], RSCA [6], RT-middleware [7], Orca [8], ROS [9], etc. Among these tools and libraries, the representative work is ROS (Robot Operating System) and its successor ROS2. ROS adopts a modular and loosely-coupled

[1] Artificial Intelligence Research Center (AIRC), National Innovation Institute of Defense Technology (NIIDT), Beijing 100166, China
[2] Tianjin Artificial Intelligence Innovation Center (TAIIC), Tianjin 300457, China
[3] State Key Laboratory of High Performance Computing, College of Computer, National University of Defense Technology, Changesha 410073, China
[*] Corresponding author: hddai@vip.163.com xjyang@nudt.edu.cn

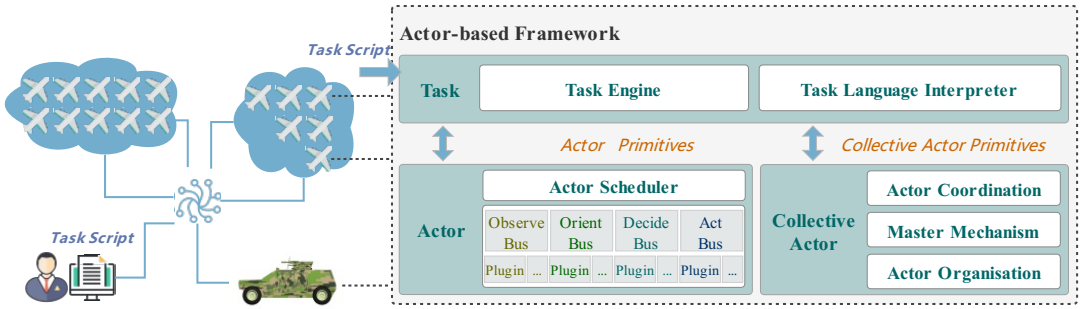



Fig. 1: The architecture overview of the Actor-based framework.

communication interface, and a light-weighted framework to support the reuse of robot drivers and algorithms. ROS2 incorporates a distributed communication middleware DDS [10] to get rid of the centralized node required by ROS. Recently, a robot operating system named 'micROS' [11], [12] was introduced for collective and cooperative robot systems based on ROS. A distributed architecture for the whole system and a layered architecture for individual nodes were proposed in 'micROS'.

Swarm robotic behaviors can be programmed in two different ways, namely the top-down and bottom-up methods. With the top-down method, a task developer is responsible for composing tasks and a centralized node is implemented to interpret the program and dispatch sub-tasks to the swarm robotic system. NVL [13] implemented a task orchestration language for composing multi-vehicle tasks. NVL enabled the dynamic selection of unmanned vehicles and the allocation of cooperative tasks, but its extensibility is limited. Dolphin [14] implemented a similar task orchestration language which provided more primitives for firing tasks. It enabled the extensibility to new robots with a flexible tool-chain. Voltron [15], which was implemented for mobile sensor networks, supported centralized control with code and data replications. Besides, there were other similar programming models that handled the coordination complexity with a central node, including Kama [16], Tecola [17] etc.

With the bottom-up method, applications are programmed from the robots' own point of view and a decentralized method is usually adopted. Individuals in the swarm robotic system can cooperate with each other based on gathered information and make decisions on their own. COROS [18] implemented a software architecture for building new distributed robotic applications on top of ROS. ALLIANCE-ROS [19] combined ALLIANCE with ROS. It aims at achieving distributed swarm robot cooperation and improving the software reusability. Buzz [20] is a domain-specific language implemented for programming applications with heterogeneous swarm robot systems. GSDF [21] is similar to Buzz but implemented in C++ language, therefore it is more compatible with available robotic libraries and tools, such as ROS and DDS. ROSBuzz [22] combined Buzz with ROS. In the bottom-up methods, collective behaviors emerge from the programmed individuals' behaviors. Therefore, it is difficult to handle tasks on-demand and reprogramming is required to cope with task modifications.

The Actor-based framework in this paper was initiated in the micROS [11] development. In the Actor-based framework, the DSL for task programming is designed for the bottom-up approach. One significant feature of the Actor-based framework, that distinguishes it from the above-mentioned bottom-up methods, is the introduction of the new control unit 'Actor', which makes a task-level reprogramming on-the-fly doable. Individuals in the swarm robotic system can switch to different Actors according to their own observation and judgment, or synchronize with others followed by a collective behavior concurrently.

## III. ARCHITECTURE OVERVIEW

Our Actor-based framework is designed to enable a task developer at the ground station to compose and fire cooperative tasks to an autonomous swarm robotic system. Fig. 1 illustrates the architecture of the Actor-based framework. The target hardware system is a heterogeneous swarm robotic system with tens of unmanned autonomous robot platforms, which communicate based on the wireless ad hoc network (WANET). All robot platforms are deployed with the proposed framework and a repository of Actor plugins beforehand. A daemon process for the Actor-based framework is launched on each platform after power-up. An opensource communication middleware FastRTPS is used for the implementation of the framework and the unreliable working mode is chosen for the wireless network.

To start a cooperative task with the autonomous swarm robotic system, a task script is broadcast from the ground station. When all platforms get the task script, the task commander can fire a *start* command. Every platform will start their initial Actors as described in the task script, and then the whole cooperative task is started. After that, if the ground station has intermittent or loss of communication with the swarm robotic system, the Actor-based framework would manage the cooperative task autonomously and handle exceptions.

## IV. DETAILS OF THE ACTOR-BASED FRAMEWORK

Our proposed framework provides a robust self-organization mechanism for swarm robotic systems and makes it convenient for users to achieve one-to-many autonomous vehicles manipulation.

### A. *Actor–The control unit for the framework*

The designed Actor is the virtualization of robot capabilities, and is the core of the proposed framework. The framework manages the whole robot swarm based on the Actor. In other words, the Actor is the basic unit for the management of swarm behaviors in our proposed framework.

**1. Actor Control Block (ACB).**

For each Actor, the Actor scheduler maintains a data structure, which is called Actor Control Block (ACB) in our software framework. The ACB encapsulates basic attributes and operations for Actors, where basic Actor attributes include the name, identification (ID), status, priority, software resources, hardware resources, permission, task, the relation with other Actors in the swarm, and the statistic information at runtime, and basic Actor operations include *start*, *stop*, *pause*, *activate*, and *switch*.

We divide the swarm robotic behaviors into four categories: swarm observation, swarm orientation, swarm decision and swarm action (OODA), then design the corresponding observation bus, orientation bus, decision bus, and action bus. Algorithm developers can design different plug-ins that mounted on different buses according to their functions. As the basic control unit, Actors are bounded to different plug-in groups, thus having different algorithms and capabilities.

**2. Some basic Actor operations.**

Based on the proposed Actor mechanism, we further design several basic operations for Actors. Fig. 2 gives the basic Actor operations and their corresponding Actor state changes. The basic operations are defined as follows:

Actor *start* operation: Load the corresponding plug-in groups, complete the binding of Actors to corresponding robotic platforms, sensors and other hardware, and complete resource allocation and algorithm loading.

Actor *stop* operation: Unlink the binding of Actors to corresponding robotic platforms, sensors and other hardware, terminate and unload the corresponding plug-in groups.

Actor *pause* operation: Achieve secure suspension and data protection of the corresponding plug-in groups, and make the Actor enter the sleeping (blocking) state from the running state.

Actor *activate* operation: Awake the sleeping Actor's plug-in groups, make the Actor enter the running state from the sleeping (blocking) state, and wait for the scheduling of the proposed Actor-based architecture.

Actor *switch* operation: Perform the *pause* operation of current Actor, load new plug-in groups, and complete the binding of Actors to new robotic platforms, sensors and other hardware.

**3. The advantage of the Actor mechanism.**

The proposed Actor achieves the conversion from infrastructure resources to robotic capabilities. A single robot platform may have resources that can support multiple Actors, so it can also run multiple Actors. Different Actors occupy different resources, adopt different algorithms, and ultimately demonstrate different abilities. The Actor-based framework enables robotic platforms autonomously switch and execute Actors according to dynamic changes of the environment and the swarm state, thereby achieving adaptive scheduling of swarm robotic behaviors.

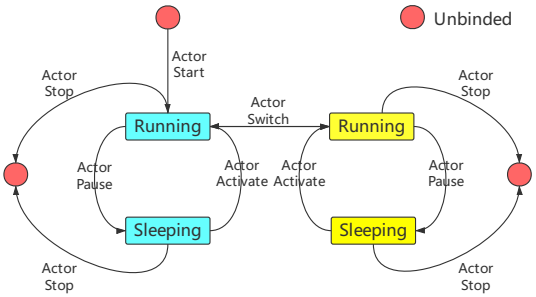



Fig. 2: Basic Actor operations and the corresponding Actor state changes.

### B. *Collective Actor*

Our framework provides a collective Actor mechanism to organize Actors in the swarm robotic system. The collective Actor mechanism in background maintains the organization information of the swarm, resolves the conflict of interest in the organization, and provides primitives for high-level coordination of collective behaviors. The mechanism also features in tolerance and supports the swarm robotic system to re-organize autonomously up against platforms' crash, communication interruption and other failure situations. Based on the introduction of the Actor organization mechanism, we achieve collective Actor management in the following three aspects.

**1. Actor organization maintenance**

Actor organization maintenance is fundamental to other collective management issues. In the Actor-based framework, each platform in the swarm periodically broadcasts a heartbeat message that includes a unique platform ID and a flag. The flag indicates whether the platform is the current Master node. Member joining/leaving events are generated based on the heartbeat reception. We have designed an interface for incorporating various heartbeat processing algorithms. If the number of heartbeat messages of a platform reaches over $M$ within $T_{join}$ seconds, it is considered joining; and if no heartbeat message was detected in $T_{leave}$ seconds, it is considered leaving. The maximum expiration time for $T_{leave}$ and $T_{join}$ can be configured by the user or calculated by heuristic methods, e.g. the accrual failure detector [23].

**2. Master election and control message delivery**

The Actor-based framework adopts a centralized-decentralized-combined scheme for the management of collective Actors. A Master is always elected among the platforms using a distributed method. The Master is responsible for organizing Actors and arbitrating when conflicts or competitions arise in the swarm robotic system. For example, two platforms compete to become Actor $A$, however, only one Actor $A$ is required in the swarm. In this case, the Master will be responsible for the final decision.

The Master handles requests from all Actors in the swarm and publishes messages to manage collective robot behaviors. Based on the communication middleware FastRTPS, we further implement reliable communication at the application

layer. The Master maintains several publishing queues to buffer control messages, and each queue publishes messages to one Actor. A message at the head of the queue will be retransmitted until receiving a reply message from the destination. Similarly, other Actors also maintain a publishing queue individually, allowing the buffering of control messages to the Master. To avoid network congestion, a linear regression scheme is adopted by increasing the delay time for retransmission linearly.

The master node in the swarm could encounter failure in challenging environments. In this situation, a new Master will be selected in a distributed manner when other nodes find the missing of the Master's heartbeat. In the proposed Actor-based framework, the platform with the minimum platform ID will be directly appointed as the Master when platforms join/leave the swarm, followed by a handshake process between the Master and other members. There are many distributed consensus algorithms such as Paxos [24], that can generally be used for the Master election. However, these algorithms are at relatively high communication cost, thus the election would be divergent and may not be finished timely when the communication is unstable.

**3. Actor coordination**

The collective Actor implements synchronization primitives for the workflow control of swarm robotic systems. An Actor-level synchronization barrier enables multiple Actors to wait until all Actors have reached a particular point of execution before any Actor continues. The synchronization barrier is recognized as a typical primitive for cooperative tasks with a swarm robotic system. All basic Actor operations, including activate, pause, stop and switch, can be synchronized. The Actor-level synchronization barrier is achieved naturally in such a way that the Master gathers barrier requests from all selected Actors and sends responses immediately after a sufficient number of requests are received. In addition, additional data can be attached in the request and response messages to support synchronization gather and scatter primitives. For example, given a sequence of integers from 1 to $N$, with $N$ indicates the number of Actors in the synchronization group, the Master can distribute this sequence among all Actors, one for each in the group. The above primitives are the basics for implementing collective behaviors at the task level.

Additionally, the collective Actor provides a mechanism to respond Actor join/leave event proactively. For the leader-follower formation algorithm, the collective Actor provides a mechanism to autonomously resolve unusual situations, such as multiple-leaders or leader lost due to platform failures.

### C. *The interaction language for human and swarm robots*

In order to provide convenience for both professional and non-professional users to compose swarm robotic tasks, we design and develop a novel domain-specific language for the interaction between human and swarm robots. As shown in Fig. 3, with the introduction of our interaction language, the task developers for robot swarms do not need to pay much attention to programming details and can focus more

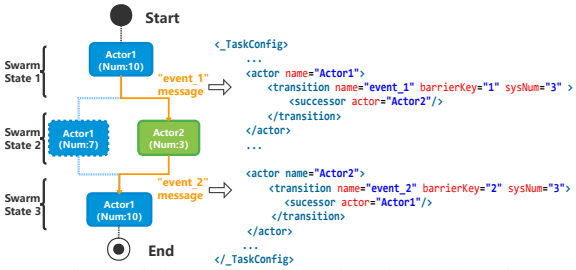



Fig. 3: A simple illustration of the developed domain-specific language. The task commander or swarm members can participate in the behavior management of swarm robotic systems directly through the 'Event' message. The 'Event'-based Actor transition (also called Actor state machine) is programmed with our designed DSL in the task script.

```
<_Task>
    <_P_Name>TaskA</_P_Name>
    <_P_TaskPrio>10</_P_TaskPrio>
    <_ActorsConfig> <!—Actors configuration module -->
        <!-- …… -->
    </_ActorsConfig>
    <_TaskConfig> <!—Task configuration module -->
        <!-- …… -->
    </_TaskConfig>
    <_PlatformActorConfig> <!—Actors initialization module -->
        <!-- …… -->
    </_PlatformActorConfig>
</_Task>
```

Fig. 4: The structure of the task script. The script is written in XML, and it mainly consists of three parts: 1) <*_ActorsConfig*> which configures the Actor information including Actor name, Actor ID, Actor priority, plug-in information, etc; 2) <*_TaskConfig*> which configures the Actor transition information (namely 'Event' module); 3) <*_PlatformActorConfig*> which configures the initial Actor information for platforms in the swarm.

on specific swarm tasks. Each platform in the swarm runs the proposed architecture and loads the same task script. In this way, all platforms can automatically manage the behaviors according to the task script. The interaction language is mainly designed for task developers.

The task script is described using our developed language and is actually an Actor state machine. As shown in Fig. 4, it is written in XML, and the user can use it to configure the detailed Actor information for robot swarms (<*_ActorsConfig*>), the Actor transition information for the task (<*_TaskConfig*>), and the initial Actor information for platforms (<*_PlatformActorConfig*>).

**1. Actor configuration in the task script**

The 'Actor' configuration is mainly used to configure the Actor information which includes Actor name, Actor priority, plug-ins required by the Actor. Task developers can design the configuration of necessary Actors in robot swarms and choose corresponding plug-ins for them. By loading different plug-ins, different Actors will be given different capabilities. In general, one Actor will have one or more plug-ins that belong to different buses (observation bus, orientation bus, decision bus and action bus). Fig. 5 gives an example for the configuration of Actor *A* and Actor *B*. Actor *A* will load *plugin_a*1 and *plugin_a*2 which belong to *observe_bus*, and *plugin_a*3 and *plugin_a*4 which belong to *act_bus*. Actor *B* will load *plugin_b*1 and *plugin_b*2 which belong to *act_bus*.

```
<_ActorsConfig>
    <Actor>
        <_P_Name>A</_P_Name>
        <_P_Prio>0</_P_Prio>
        <OODAConfig>
            <Bus>
                <_P_Name>observe_bus</_P_Name>
                <Plugin>
                    <_P_Name>plugin_a1</_P_Name>
                    <_P_Name>plugin_a2</_P_Name>
                </Plugin>
            </Bus>
            <Bus>
                <_P_Name>act_bus</_P_Name>
                <Plugin>
                    <_P_Name>plugin_a3</_P_Name>
                    <_P_Name>plugin_a4</_P_Name>
                </Plugin>
            </Bus>
        </OODAConfig>
    </Actor>
    <Actor>
        <_P_Name>B</_P_Name>
        <_P_Prio>0</_P_Prio>
        <OODAConfig>
            <Bus>
                <_P_Name>act_bus</_P_Name>
                <Plugin>
                    <_P_Name>plugin_b1</_P_Name>
                    <_P_Name>plugin_b2</_P_Name>
                </Plugin>
            </Bus>
        </OODAConfig>
    </Actor>
</_ActorsConfig>
```

Fig. 5: An example of the Actor information configuration (<*_ActorsConfig*> module) in the task script.

```
<actor name="A">
    <transition name="AtoB_event" sysNum="10" timeout="60">
        <successor actor="B" /> <!-- switch to Actor B -->
        <timeoutBranch> <!-- timeoutBranch not compulsory -->
            <successor actor="C" /> <!-- switch to Actor C -->
        </timeoutBranch>
    </transition>
    <param name="TargetPoint" value="200 200 20"/>
    <transition name="AtoD_event">
        <successor actor="D" />  <!-- switch to Actor D -->
    </transition>
    <transition name="pause_event">
        <successor actor="NULL" />  <!-- pause Actor A  -->
    </transition>
</actor>
```

Fig. 6: An example of the 'Event' processing module for Actor transition description in the task script.

**2. Event-based Actor transition**

The task configuration mainly consists of several 'Event' processing description modules. The conditions of the Actor transition are described and set in the 'Event' processing description modules. In other words, the 'Event' module is designed for the Actor transition in the interaction language. In this way, task implementation is achieved through the trigger of event messages based on our Actor-based framework. The configuration for 'Event' module includes *name*, *sysNum* (the number of triggered Actors), *timeout*, *param* (Actor parameters), *barrierKey*, etc. Fig. 6 shows an example of an 'Event'-based module for the description of Actor transition.

*sysNum* is the number of Actors required to transition when receiving the corresponding 'Event' message. *timeout* is the time limit for the synchronized transition. In the example in Fig. 6, when receiving the *AtoB_event* message, the task engine for the Actor-based framework will arrange 10 robots to switch from Actor *A* to Actor *B*. The task engine will wait for 60 seconds for robot swarms to negotiate and respond with candidates for the transition. If a robot cannot get the response successfully in 60 seconds, the task engine for the Actor-based framework will go into *timeoutBranch* and force the robot to switch from Actor *A* to Actor *C*. *param* is a set of key-value pairs. In the case shown in Fig. 6, we set the parameter *TargetPoint* to '200 200 20' for Actor *D*.

```
<actor name="A">
    <transition name="AtoBC_branch_event" barrierKey="4"
sysNum="10">
        <branch cnt="2">
            <successor actor="B" />
        </branch>
        <branch cnt="8">
            <successor actor="C" />
        </branch>
    </transition>
</actor>
<actor name="B">
    <transition name="finish_event" barrierKey="5" sysNum="2">
        <branch cnt="2">
            <successor actor="A" />
        </branch>
    </transition>
</actor>
<actor name="C">
    <transition name="finish_event" barrierKey="5" sysNum="8">
        <branch cnt="8">
            <successor actor="A" />
        </branch>
    </transition>
</actor>
```

Fig. 7: An example of the 'Event' processing module. It shows the usage of *barrierKey* in branching and aggregation for collective Actors in the task script.

**3. Barrier mechanism for swarm behavior consistency**

Our task orchestration language provides a barrier mechanism that the swarm will wait until there are a sufficient number of Actors ready to switch their state (Actor transition). The user can achieve this by configuring *barrierKey*. *barrierKey* is a state synchronization key. It is a global synchronization identifier of swarm robotic systems. It supports the simultaneous transitions of different Actors. The value range is 1-63. The *barrierKey* can be set according to the demand. When the same *barrierKey* is set for multiple 'Event' messages, the task engine will wait at the synchronization point until all corresponding event messages are generated and a sufficient number of Actors are ready to switch. This synchronization mechanism enables the consistency of swarm behaviors. As shown in Fig. 7, we set the same *barrierKey* in *finish_event* module for both Actor *A* and Actor *B*. When receiving the *finish_event*, 2 Actor *B*s and 8 Actor *C*s will switch to 10 Actor *A*s at the same time according to the setting of *barrierKey* (both are set to 5).

**4. Branching and aggregation of collective Actors**

We also provide the branching and aggregation operations for robot swarms in our designed interactive language. The branching mechanism supports the robot swarms to be split into multiple robot sub-swarms to perform different sub-tasks. The aggregation mechanism supports the re-aggregation of multiple robot sub-swarms to be one robot swarm after finishing sub-tasks in order to continue the task. When the robot swarm is divided into multiple sub-swarms, the proposed collective Actor organization and management mechanism will perform Actor sub-swarm management according to the status of each member in the sub-group to coordinate different collective tasks. Fig. 7 gives an example of how to set branching and aggregation in task scripts. As shown in the figure, *barrierkey* is used both in branching and aggregation. When receiving *AtoBC_branch_event* message, the swarm (10 Actor *A*s) will branch into two sub-swarms (2 Actor *B*s and 8 Actor *C*s, separately). When receiving *finish_event* message which sets the same *barrierkey*, the


(a) (b) (c)

Fig. 8: Platforms used for the experimental evaluation. (a) The customized module for hardware-in-loop simulation. (b) The quadrotor drones used in the in-field experiments. (c) The fixed-wing UAV model used in the simulation.

sub-swarm will aggregate into one swarm.

**5. External commands from users or ground station to manipulate robot swarms**

During task execution, based on the observation and perception by swarm members (position, power, status, etc.) and the external environment (objects, obstacle, weather, etc.), the Actor can make corresponding decisions and generate corresponding event messages. Then the robots in the swarm will behave either individually or collectively through Actor transition according to the task script.

Besides the autonomous Event message generating by the robot swarm, we also provide the external control interface for human-swarm interaction at runtime. The task commander can send 'Event' messages to the robot swarm from the ground station, thereby achieving external manipulation of the robot swarm. The 'Event' messages from the ground station are interpreted as an urgent intervention from the task commander, thus being given a higher priority of processing than that from the robot swarm. After receiving the 'Event' message, the robot swarm will execute immediately, following operations assigned for Actors such as forced transition, forced starting, forced stopping, data backup, etc.

## V. Experimental Evaluation

The Actor-based framework is running on micROS. We test it on both ARM-based machines and X86-based machines. Quantitative evaluation is firstly performed to prove the efficiency of our proposed Actor-based framework in swarm organization and management. Then, 30 fixed-wing UAVs in the Gazebo simulation environment are adopted to demonstrate the ability of the framework. Finally, 10 quadrotor drones are used for conducting the experimental evaluation in the field.

### A. *Quantitative evaluation of the framework*

We firstly perform the quantitative evaluation of the framework based on the customized module. The customized module is shown in Fig. 8 (a). It consists of an ODROID-XU4, a 1W miniature OEM 2.4GHz Ethernet/Serial/WIFI Router (PX2) produced by MicroHard, and a 915MHz communication module. The ODROID-XU4 includes an ARMv7 CPU with 8 cores and 2GB memory. We test the overload of the framework with the case that two Actors alternate with each other at a frequency of 1 Hz. The test is repeated ten times. As shown in table I, the CPU and memory overload of the framework on ODROID-XU4 are 0.25% and 1.30% respectively. The mean Actor switching time is 104ms.

TABLE I: The overload of the Actor-based framework running on ODROID-XU4 and TX2.

| Platform | CPU Usage | Memory Usage | Actor Switching Time (ms) |
|---|---|---|---|
| ODROID-XU4 | 0.25% | 1.30% | 104 |
| TX2 | 1.11% | 0.40% | 105 |

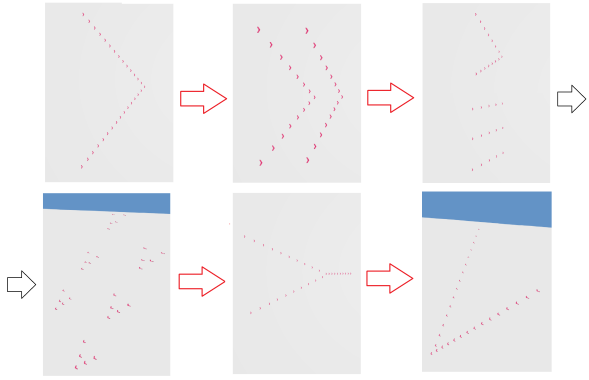


Fig. 9: The computational simulation with 30 fixed-wing UAVs. All UAVs are equipped with our proposed framework.

Then we use the customized drones (as shown in Fig. 8 (b)) which are equipped with NVIDIA Jetson TX2, a 915MHz communication module and a 5.8GHz communication module to test the framework. The NVIDIA Jetson TX2 module integrates a 256 core NVIDIA Pascal GPU, an ARMv8 Multi-Processor CPU Complex (including a dual-core NVIDIA Denver 2 and a quad-core ARM Cortex-A57), and a DRAM with 8GB memory. As shown in table I, for running our Actor-based framework, the CPU and memory overloads are 1.11% and 0.40% respectively. The mean Actor switching time is 105ms.

We evaluate the performance for collective Actor management in three scenarios, including collective Actor switching with the barrier, collective Actor branching with the barrier, and new Master election.

Firstly, for the collective Actor switching case, the detailed communication cost is given in Table II(a). For the collective Actor switching operation, Actors in the swarm need to interact with each other to confirm the consistency of Actor state transition. The experimental results show that the communication duration extends as the number of Actors in the swarm increases. In addition, we study statistically the total number of inlet and outlet packages for different nodes. The network traffic of the Master node is much higher than that of non-Master nodes. We repeat each test ten times to reduce the impact of the communication environment. Table II(a) also shows that the number of Actor plugins has little impact on the communication duration and network traffic.

Secondly, for the collective Actor branching case, the detailed communication cost is given in Table II(b). In the experiment, two Actors are randomly selected from the swarm to perform the branching operation. As shown in the table, the interaction duration for completing a branching operation also increases with the increment of the swarm size. Since only two Actors are selected for the synchronized transition, the duration is much shorter compared with that of

TABLE II: The quantitative evaluation of the collective Actor transition with our designed Actor-based framework. The framework runs on TX2, deployed with the case (a) that all Actors in the swarm switch synchronously on the barrier mechanism, and the case (b) that two Actors are randomly selected from the swarm to perform the branching operation. For all the data in the table, we repeat the experiment ten times and take an average. The package is about 158.6 Bytes/pkg.

| Swarm Size | (a) Actor Switching(one plugin / five plugins) | | | (b) Actor Branching(one plugin / five plugins) | | |
|---|---|---|---|---|---|---|
| | transition time(ms) | network traffic(pkg/s) (Master) | network traffic(pkg/s) (non-Master) | transition time(ms) | network traffic(pkg/s) (Master) | network traffic (non-Master) |
| 5 | 352 / 449 | 48 / 54 | 35 / 37 | 206 / 230 | 48 / 52 | 34 / 35 |
| 10 | 516 / 488 | 156 / 182 | 73 / 68 | 268 / 239 | 198 / 189 | 67 / 70 |
| 15 | 546 / 558 | 435 / 415 | 114 / 120 | 279 / 277 | 441 / 427 | 103 / 109 |
| 20 | 588 / 649 | 575 / 798 | 168 / 149 | 418 / 278 | 770 / 626 | 144 / 128 |
| 25 | 1246 / 1127 | 1156 / 1008 | 178 / 195 | 429 / 443 | 1216 / 1160 | 179 / 170 |
| 30 | 1332 / 1326 | 1635 / 1509 | 220 / 194 | 529 / 458 | 1629 / 1704 | 209 / 197 |

TABLE III: The time for a new Master to be elected from the swarm after the origin Master fails. We repeat the experiment ten times and take an average.

| | Swarm Size | | |
|---|---|---|---|
| | 10 | 20 | 30 |
| Election Time (ms) | 382 | 414 | 909 |

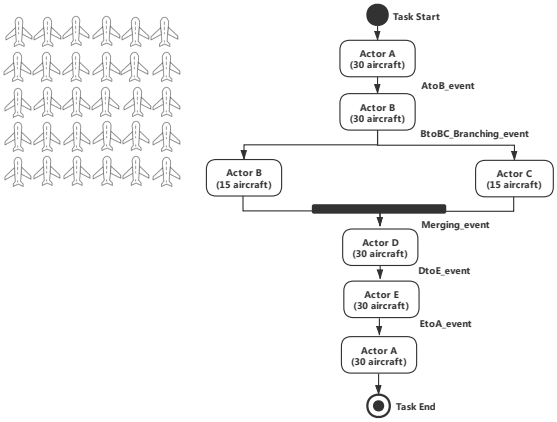



Fig. 10: The Actor state machine for the simulation experiments with our framework.

the case in Table II(a). The Master node also suffers from a heavier burden on network traffic than that with non-Master nodes.

Thirdly, in order to test the robustness of our proposed framework, we measure the time required to re-elect a new Master node. In the test, we kill the daemon process on the Master node and measure the recovering time when a new Master is elected from the swarm. As shown in Table III, the election time increases with the increment of the swarm size, and the duration ranges from 382ms to 909ms.

### *B. Qualitative evaluation based on computational simulation*

We also evaluate our proposed framework through computational simulation. We used 30 fixed-wing UAVs in Gazebo to test the framework. Note that the Actor-based framework is compatible with ROS, so we can use Gazebo directly to set up the simulation environment. Fig. 8(c) shows the model of the fixed-wing UAV in the simulation.

As shown in Fig. 9, the task commander can easily achieve one-to-many fixed-wing UAV manipulation. 30 fixed-wing UAVs in the swarm all start with Actor *A* in the beginning,

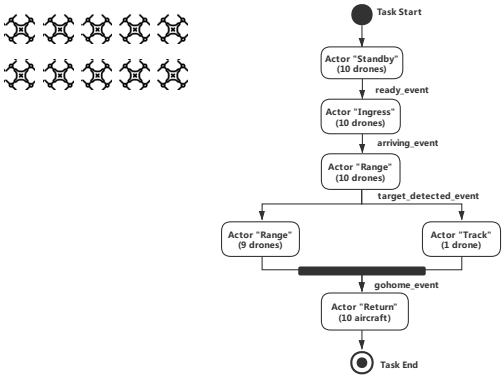


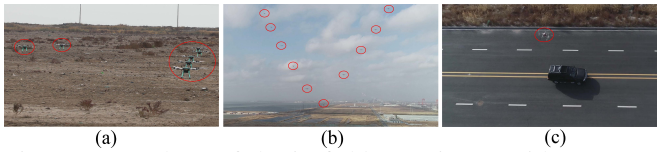

Fig. 11: The Actor state machine for the in-field experiments with our framework.

and the swarm keeps one arrow formation for a long-distance flight. After a while, when receiving the *AtoB_event* message, all UAVs switch from Actor *A* to Actor *B*, and the swarm switches to two arrow formation. Then, the swarm receives a *BtoBC_branching_event* message and performs the branching operation. 15 UAVs switch from Actor *B*s to Actor *C*s (linear shape formation). Afterward, the swarm (15 Actor *B*s and 15 Actor *C*s) switches to Actor *D*s which keeps six arrow formations. Following that, all UAVs switch from Actor *D*s to Actor *E*s, and keeps herringbone formation. In the end, the swarm switches to Actor A again to return. The Actor state machine for this task is shown in Fig. 10.

### *C. Qualitative evaluation based on in-field swarm test*

We also achieve the search-and-track task with a swarm of 10 quadrotor drones, which are deployed with the Actor-based framework. When the quadrotor drones are ready at their take-off positions, we broadcast the task script to the swarm from the ground station. After all swarm members confirm the reception of task scripts, a *start* command is fired from the ground station. The swarm starts up running the Actor state machine, as is shown in Fig. 11. Then the swarm carries out the complex task autonomously, and intervention from the ground station is no longer necessary except for the recycling of the swarm.

Snapshots of the in-field experiment are shown in Fig. 12. 10 drones all start with Actor *Standby*, taking off to the specified altitude. When an Actor *Standby* arrives at the target altitude, it autonomously send a *ready_event* message.



(a) (b) (c)

Fig. 12: Snapshots of the in-field experiment with a swarm of 10 quadrotor drones. The drones in all snapshots are highlighted with circles around them. (a) Task deployment; (b) Actor *Ingress*; (c) Actor *Track*.

The framework would confirm that all Actor *Standby* are ready, and then simultaneously enable the switching to Actor *Ingress*, with which 10 drones fly toward the target searching area in an arrow formation. Similarly, when an Actor *Ingress* arrives at the target area, the Actor publishes an *arriving_event* message. 10 drones synchronously switch from Actor *Ingress* to Actor *Range*. Actor *Range* is responsible for seeking out the target, e.g. a jeep in this experiment. A *target_detected_event* message is published by the Actor *Range* once it detects the target. According to the task script, one of Actor *Range*s is then chosen to switch to Actor *Track*, and begins to follow the target. Finally, we fire a *go_home_event* message from the ground station. All drones switch to Actor *Return*, and return to the landing area.

## VI. CONCLUSIONS AND FUTURE WORK

We present an Actor-based framework for the programming of autonomous swarm robotic systems. The introduction of the control unit 'Actor' helps decouple the high-level task programming with specific robot platforms. With the proposed framework, the swarm robotic system can autonomously handle unusual situations, such as node failures, network disturbances, etc., and continue the deployed task. The framework is quantitatively and qualitatively validated to prove its efficiency. In the future, we plan to build up an Actor factory that incorporates Actors and the corresponding sets of plugins for typical industrial and academic scenarios.

## ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (Grant No. 91948303). The authors would like to thank Chunlian Fu, Junling Gao for their help in the implementation of the framework, and Dr. Ying Chen for her help in polishing the writing of the paper.

## REFERENCES

[1] M. Dadgar, S. Jafari, and A. Hamzeh, "A PSO-based multi-robot cooperation method for target searching in unknown environments," *Neurocomputing*, vol. 177, pp. 62 – 74, 2016.

[2] H. Sugiyama, T. Tsujioka, and M. Murata, "Real-time exploration of a multi-robot rescue system in disaster areas," *Advanced Robotics*, vol. 27, no. 17, pp. 1313–1323, 2013.

[3] A. C. Kapoutsis, S. A. Chatzichristofis, L. Doitsidis, J. B. de Sousa, J. Pinto, J. Braga, and E. B. Kosmatopoulos, "Real-time adaptive multi-robot exploration with application to underwater map construction," *Autonomous Robots*, vol. 40, no. 6, pp. 987–1015, Aug 2016.

[4] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, "Miro-middleware for mobile robot applications," *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 493–497, 2002.

[5] H. Bruyninckx, "Open robot control software: The OROCOS project," in *In 2001 IEEE International Conference on Robotics and Automation (ICRA)*, vol. 3. IEEE, 2001, pp. 2523–2528.

[6] J. Lee, J. Park, S. Han, and S. Hong, "RSCA: Middleware supporting dynamic reconfiguration of embedded software on the distributed URC robot platform," in *The 1st International Conference on Ubiquitous Robots and Ambient Intelligence*. Citeseer, 2004, pp. 426–437.

[7] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, "RT-middleware: distributed component middleware for RT (robot technology)," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2005, pp. 3933–3938.

[8] A. Makarenko and A. Brooks, "Orca: Components for robotics," in *In 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Citeseer, 2006.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[10] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[11] X. Yang, H. Dai, X. Yi, Y. Wang, S. Yang, B. Zhang, Z. Wang, Y. Zhou, and X. Peng, "micROS: A morphable, intelligent and collective robot operating system," *Robotics and Biomimetics*, vol. 3, no. 1, p. 21, 2016.

[12] H. Dai, X. Yi, Y. Wang, Z. Wang, and X. Yang, "Parallel learning architecture of micros powering the ability of life-long autonomous learning," *Journal of Computer Research and Development*, vol. 56, no. 1, pp. 49–57, 2019.

[13] E. R. Marques, M. Ribeiro, J. Pinto, J. B. Sousa, and F. Martins, "NVL: A coordination language for unmanned vehicle networks," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 2015, pp. 331–334.

[14] K. Lima, E. R. Marques, J. Pinto, and J. B. Sousa, "Dolphin: A task orchestration language for autonomous vehicle networks," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 603–610.

[15] L. Mottola, M. Moretta, K. Whitehouse, and C. Ghezzi, "Team-level programming of drone sensor networks," in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. ACM, 2014, pp. 177–190.

[16] K. Dantu, B. Kate, J. Waterman, P. Bailis, and M. Welsh, "Programming micro-aerial vehicle swarms with Karma," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2011, pp. 121–134.

[17] M. Koutsoubelias and S. Lalis, "TeCoLa: A programming framework for dynamic and heterogeneous robotic teams," in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ACM, 2016, pp. 115–124.

[18] A. Koubâa, M.-F. Sriti, H. Bennaceur, A. Ammar, Y. Javed, M. Alajlan, N. Al-Elaiwi, M. Tounsi, and E. Shakshuki, "Coros: A multi-agent software architecture for cooperative and autonomous service robots," in *Cooperative Robots and Sensor Networks 2015*. Springer, 2015, pp. 3–30.

[19] M. Li, Z. Cai, X. Yi, Z. Wang, Y. Wang, Y. Zhang, and X. Yang, "ALLIANCE-ROS: A software architecture on ROS for fault-tolerant cooperative multi-robot systems," in *Pacific Rim International Conference on Artificial Intelligence*. Springer, 2016, pp. 233–242.

[20] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 3794–3800.

[21] X. Chang, Z. Cai, Y. Wang, X. Yi, and N. Xiao, "GSDF: A generic development framework for swarm robotics," in *International Conference on Intelligent Robotics and Applications*. Springer, 2017, pp. 659–670.

[22] D. St-Onge, V. S. Varadharajan, G. Li, I. Svogor, and G. Beltrame, "ROS and Buzz: Consensus-based behaviors for heterogeneous teams," *arXiv preprint arXiv:1710.08843*, 2017.

[23] R. Y. N. Hayashibara, X. Defago and T. Katayama, "The /spl phi/ accrual failure detector," in *23rd IEEE International Symposium on Reliable Distributed Systems*. IEEE, 2004, pp. 66–78.

[24] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79 – 103, 2006.