Handling Failures In A Swarm

Gaurav Verma[1], Lakshay Garg[2], Mayank Mittal[3]

***Abstract*— Swarm robotics is an emerging field of robotics research which deals with the study of large groups of simple robots. Swarms can exhibit complex behaviors even when each of the member robot performs only simple actions. Since swarms involve interaction and cooperation between multiple robots, they are seen as more robust, flexible and efficient as compared to traditional single agent systems [1]. In this project, we worked on a method which allows a swarm to diagnose faults and take corrective actions autonomously.**

I. Introduction

Swarm robotics is a field of robotics research inspired by the emergent behavior of large groups of simple creatures. Swarms exhibit complex behavior unknown to individual agents. It is believed that swarms more robust, flexible and efficient as compared to traditional single agent systems but these characteristics are not something which are inherent to the system and are only achieved by careful design and robust algorithms. The potential of such systems remains largely under-utilized because of the challenges and complexities involved in multi-robot systems. Despite their challenges, they can be used to accomplish complex tasks with simple robots by collaboration and load sharing among agents and therefore are a promising field of research. If swarms are to be used in industrial and other useful applications then they must be made more reliable and tolerant to faults. Several approaches have been explored by different authors over the years. In this project, we take inspiration from the work done in field of wireless sensor networks (WSN) [2] and implement a majority voting based algorithm to detect and remedy faults in a swarm of quadrotors.

II. Problem Statement

Swarms of robots provide us with more flexibility and efficiency but can fail catastrophically if one of the agent fails. In this project we deal with the problem of ensuring that a swarm can keep functioning without any manual intervention. In particular, we consider the scenario in which a swarm (possibly heterogeneous) is assigned a task which must be completed by its agents. The swarm may suffer from failures time-to-time. We also have some other robots which can be used as substitutes to replace the ones which are working in the swarm.

[1]gverma@iitk.ac.in
[2]lakshayg@iitk.ac.in
[3]mayankm@iitk.ac.in

III. Proposed Solution

The approach we propose can be divided into the stages

1) Partitioning the task into subtasks for assignment to individual agents in the robot swarm
2) Detecting faults in swarm agents by communication between *neighbors* in the system
3) Taking remedial action to resolve fault in system

A. Partitioning The Task

The task to be performed by the swarm is divided into smaller non-overlapping tasks. We call each of these tasks a *function*. For the case of constructing a house, let the task be partitioned into N functions $f_1, f_2, \ldots, f_N$ where N is the number of agents in the swarm. Each of the member is assigned a function which it must execute. The partitioning is done by the user or by a domain specific task planner and is not related to the overall task of fault detection mechanism. This is a separate problem which involves proper sub-division of tasks in a manner such that all the required deadlines are met and the swarm does not end up in a deadlock.

B. Detecting Faults

We divide the swarm into smaller *logical neighborhoods* consisting of a small number of robots which can communicate among with each other. An example is shown in figure 1. Every robot can be a part of more than one neighborhoods. We initialize the neighborhoods in a static manner for simplicity but better methods can be devised.

Robots within a neighborhood communicate periodically with each other and send a *health signal* to all others in that neighborhood. Whenever one of the robot does not receive a health signal from its neighbor, it generates a warning message containing the robot ID and the function that it was executing and sends it to a standby robot. When the standby robot gets two or more warning messages containing the same robot ID, it concludes that the robot has failed and goes on to take its place. The substitute waits for at least two messages because it may occur that a faulty robot is erroneously

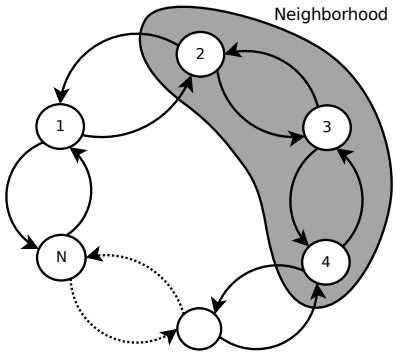

Fig. 1. An example of a logical neighborhood within the swarm

sending messages to the substitute. An example is shown in figure 2.

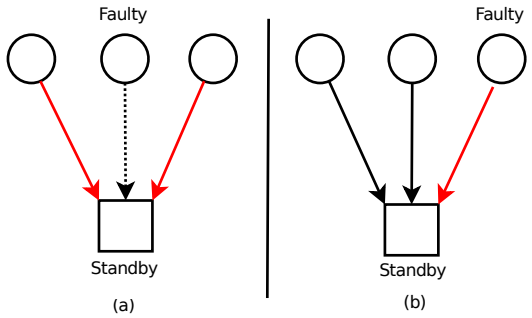

Fig. 2. Two cases of fault are shown in the figure. Red lines represent a warning message being transmitted and the dotted lines represent that a message may or may not be sent. In case (a) the standby robot takes the place of the faulty robot whereas in case (b) the shown robot does not substitute it. The substitute in this case will be called by a different neighborhood

To ensure that both neighbors send the warning message to the same substitute robot, we also define a mapping between the neighborhoods and substitutes. This is analogous to the mapping between main memory and cache in modern computer architectures. See figure 3.

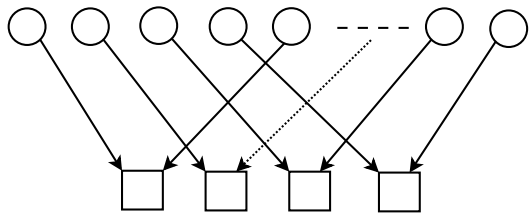

Fig. 3. Each robot is mapped to a substitute which must take its place in case of a failure

C. Resolving Faults

Once a fault has been confirmed, the substitute robot will receive the ID of failed robot and will know the task that it was performing. This information can be used by this robot to replace the failed robot. Although we are substituting failed robots, this is not the only possible strategy. We can define other remedial measures that will be taken whenever we have a failure. These remedial measures may include replanning the entire task so that functions can be reassigned or informing a base station.

IV. Implementation Details

A. State Machine

The state machine has been designed in a decoupled manner from the main robot. This has been done keeping in mind that a modular design will allow for extension to other kinds of robots and different applications without friction. The state machine can be thought of as a black-box which takes a single input (health) from the robot and gives as output a single command (task ID to perform) to the robot. The swarm can therefore be thought of as a web linking these blackboxes which are then linked to the robots (Figure 4). From now onward, we refer to the blackbox+robot system as the robot for brevity.

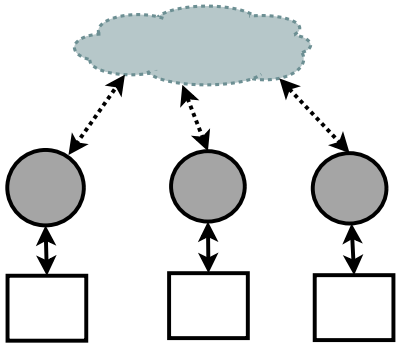

Fig. 4. The architecture of the swarm. It is a web of blackboxes which communicate by broadcasting information. The dotted lines connected to the cloud denote the blackboxes broadcasting information. Robots are connected to these blackboxes by a single link

The robots' *brain* is a 3-level hierarchical state machine where each level is made up of cascade and parallel composition of simpler state machines. Before proceeding to the actual design of the state machine, we explain how the swarm is initialized and how the robots communicate.

The swarm is initialized in a static manner in the sense that the assignment of tasks, neighborhoods and robot roles is done before the swarm can start operating. Each robot can be though of maintaining a list of variables which must be assigned before starting the swarm. A robot maintains the following variables:

1) `int robot_id`: Robot's unique identification number
2) `int neighbors[]`: Robots present in its neighborhood

3) `int init_state`: Act as a substitute or not
4) `int subs[]`: Robots which it is allowed to substitute

Robots communicate by broadcasting messages at periodic intervals, each message contains several fields. A message contains the following information:

1) `int ping_id`: The ID of robot which sent this message
2) `int warn_id`: ID of the failed robot

Every robot receives messages from every other robot and decides to listen / ignore the message based on its neighbors list. Each robot can then act on the message based on its state. A robot can also be present in a third state which occurs when it is transitioning from the inactive to the active state. The first level of the state machine is shown in figure 5

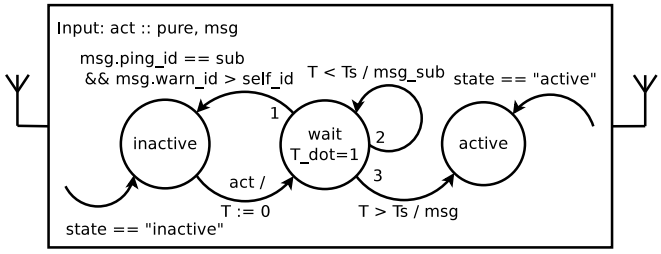

Fig. 5. Top level state diagram of the robot. The robot can be in two states: inactive, which is the state of a robot which is waiting to substitute another robot; active, the robot that is actively working as a part of the swarm.

Here, `act` is a message that is generated from a lower level in the state machine, it tells the robot to substitute a particular robot which has failed. In the active state, the robot must periodically broadcast messages and also monitor messages from its neighbors. Therefore the state `active` shown in figure 5 can be thought of as a state machine itself. The internal design of this state is shown in figure 6

In the `inactive` state, the robot is required to listen to messages about robots which it can substitute. If it detects that multiple robots say that a particular node in the swarm is faulty then it needs to respond. The state diagram for this state is shown in figure 8

The state machine shown in figure 9 has a shortcoming that it may happen that multiple substitute robots may get activated on receiving the warn signal are cause unexpected behavior. This situation can be tackled by introducing the `wait` state in top level state machine. Whenever a substitute is going to act, it first comes to the `wait` state where it broadcasts a particular kind of message with `ping_id` as the ID of failed robot and `warn_id` as its own ID. This message serves two purposes: first,

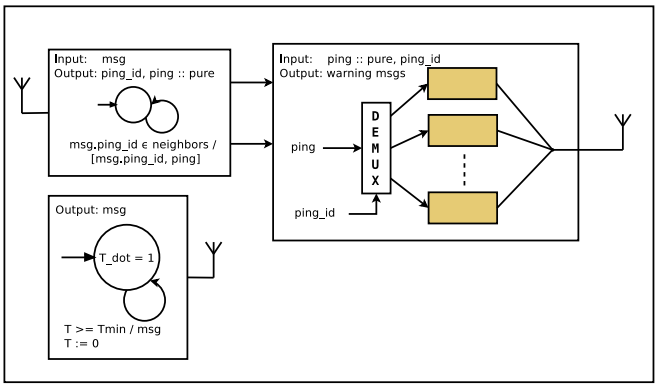

Fig. 6. Internals of the `active` state. The antennas indicate that the message is sent/received via the broadcast channel. The state machine in the lower left periodically sends message for broadcast and the cascade combination shown in the figure is the one which reads messages from neighbors. The input stage acts as a filter and ignores messages from non-neighbors. The output stage counts the number of messages each neighbor has failed to send. The yellow boxes are state machines (shown in figure 7) which maintain this count corresponding to every neighbor and generate a warning message as soon as a fault is detected.

it causes the neighbors to stop issuing warnings and second, when another robot which is trying to substitute the failed robot listens this message, it compares the `warn_id` to its `robot_id`, if the `warn_id` is smaller, it infers that another substitute has taken action and goes back to the `inactive` state.

B. ROS Implementation and Simulation

Continuing with the methodology proposed, simulation of the quadrotor as been done using Gazebo. The state machine for has been implemented in form of a ROS package `swarm_node`. The source code for this package is available on GitHub.

The quadrotor simulation has been done using the model `hector_quadrotor`. This model developed at TU Darmstadt provides a complete package for simulation of real time algorithms on quadrotors. This model has bee used for simulating the system and identifying any faults in the communication protocol.

By publishing actuation details on the topic `cmd_vel`, each of the quadrotors can be made to perform motion. The simulation so far is primitive as factors like gravity and other natural forces haven't been considered. Nevertheless, the work is sufficient to test out the communication protocol.

C. Communication

The communication between robots has been done using XBee. A XBee is a wireless transceiver used for bidirectional communication at moderate speeds. The XBee is connected to an XBee Explorer board which is used as an interface between a

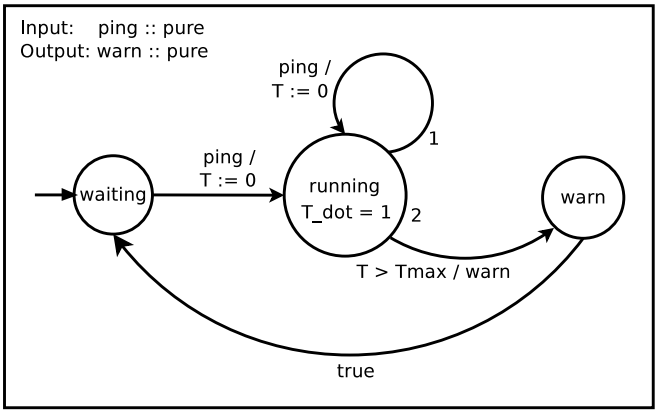

Fig. 7. This state machine is used to generate a warning message corresponding to a particular neighbor. It starts in the `waiting` state in which it waits till it receives first message from the neighbor. In the `running` state, this machine waits till `Tmax` time and generates a warning message if it has not received a message from the neighbor. It continues to generate the `warn` message till it receives a message from the substitute robot. The numbers written on transition arrows denote their priority, if both the transitions are possible then we do them in the priority order specified by these numbers. Transition 1 is taken first, after which the guards are re-evaluated and then transition is made

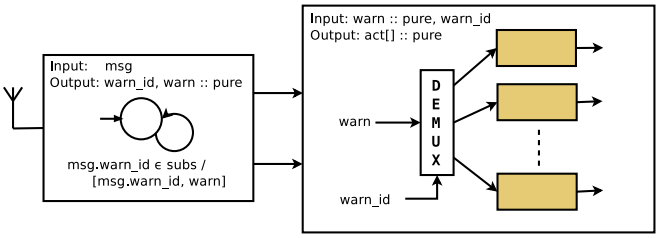

Fig. 8. The state diagram for `inactive` state is a cascade of two simpler state machines. The input stage is a filter which discards messages which are not relevant to this robot. The output stage listens if a warning message is present and for for which node. The yellow blocks shown are the state machines which act according to the warning message. The design of these blocks is shown in figure 9

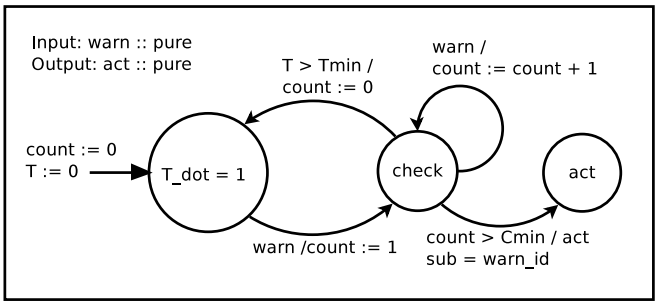

Fig. 9. This state machine counts the warning messages it receives corresponding to a particular robot. If the count is greater than a certain minimum number within some time window, it considers that the node is faulty and issues an `act` message which causes the robot to transition from `inactive` to `active` state. The numbers in transition arrows denote their priority are used to decide which transition to take in case multiple guards are true.

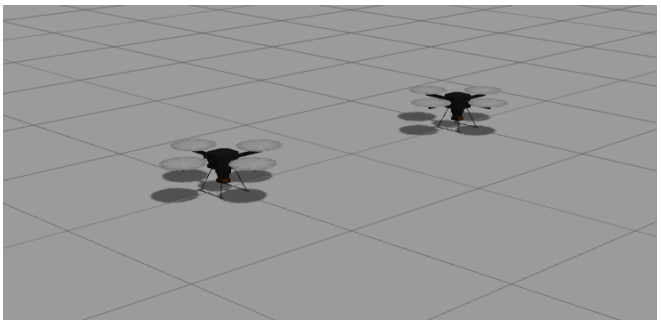

Fig. 10. Two hector quads spawned into the scene in Gazebo. Using different namespaces any number of quadrotors with its own controllers and sensors can be simulated.

computer/microcontroller and the XBee. XBee Explorer can be used to setup and configure a P2P communication network among two or more XBees.

We have studied the working and setup of XBees and the same is presented in the rest of this section.

1) Peer to Peer Communication: To configure a P2P communication network between 2 XBees it is important to understand the three levels involved in their networking:

Channel: This level controls the frequency band that the XBee communicates over. Most XBees operate on the 2.4GHz, 802.15.4 band, and the channel further calibrates the operating frequency within that band.

Personal Area Network ID (PAN ID): Two XBees can communicate with each other, only if their PAN ID is same. PAN ID is a hexadecimal number ranging between 0x0 and 0xffff. Since there are 65536 possibilities, there are miniscule chances of faulty/undesirable communication.

My Address and Destination Address: Each XBee must be assigned a source address which is called MY address and a destination address. Both the addresses are between `0x0000` and `0xFFFF`. For one XBee to be able to send data to another, it must have the same destination address as the other XBees source. For example, if XBee 1 has a MY address of `0x1234`, and XBee 2 has an equivalent destination address of `0x1234`, then XBee 2 can send data to XBee 1. But if XBee 2 has a MY address of `0x5201`, and XBee 1 has a destination address of `0x5200`, then XBee 1 cannot send data to XBee 2. In this case, only one-way communication is enabled between the two XBees (only XBee 2 can send data to XBee 1).

2) Broadcast Mode Communication: It is important to take note of the fact that broadcast mode communication using XBee requires one Coordinator and other Router/End Devices. The necessity of having a coordinator, does not go hand in hand with the scalability that we are trying to achieve, but nonetheless a XBee

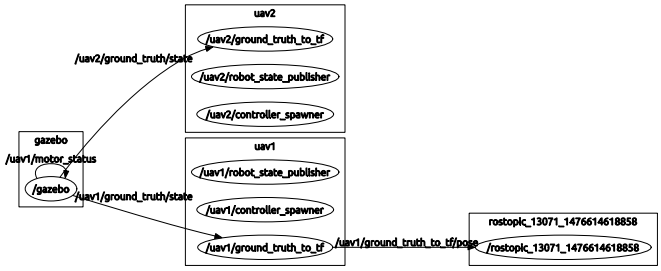

Fig. 11. This is the ROS node graph for simulation

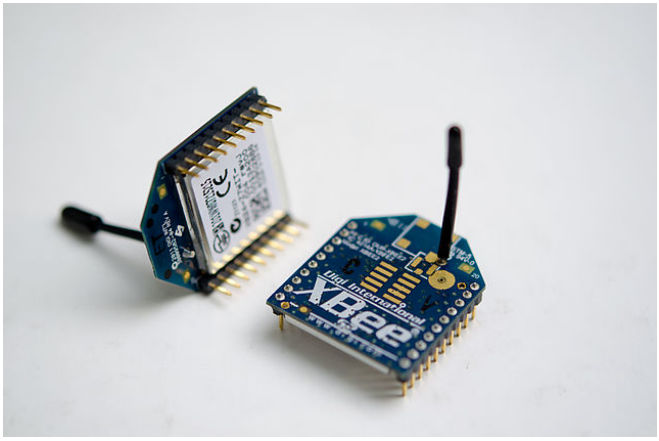

Fig. 12. A pair of XBees Series 2S. Source: Mark Fickett

network can be established to demonstrate the concept. A telemetry kit would have been a better alternative(as far as scalability of the network is considered). The networking parameters for broadcast mode communication are

PAN ID for the entire network	607
Dest. Address (High) for Coordinator	0x0000
Dest. Address (Low) for Coordinator	0xFFFF
Dest. Address(High) for Router	0x0000
Dest. Address(Low) for Router	0x0000

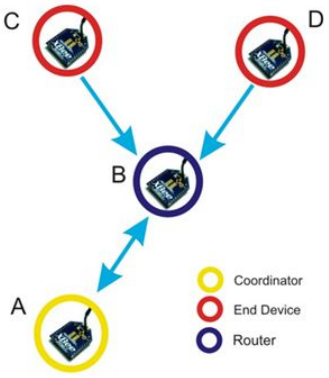

Fig. 13. Schematic of communication using XBees. Source: XBee Wikispaces

After successfully establishing the network, these XBees can share the data with other XBees in the same network. Example: A temperature sensor connected to the Arduino board senses the temperature and the Arduino instructs the connected XBee module to broadcast the sensor data to other XBees in the network that are in the same neighborhood. Other XBees (in the same neighborhood), on receiving the data, pass it onto the Arduino boards connected to them. The received sensor data (i.e. temperature value) can be analyzed to check if it is well within the pre-declared limits. If it is not, the Arduino detects the anomaly and instructs the XBee connected to it to generate a fault signal. If more than one fault signal is received from the same neighborhood, it implies that there has been a failure in that neighborhood. And now the base station can take up the task of substituting the failed robot and the group membership lists can be updated.

V. Experimental Results

To peform any experiments on the state machine, we needed to define other state machines such as the environment and the robot which would then be a complete system model and can be simulated. The models we used for the environment and the robot were very simple ones. The environment model is shown in the figure below

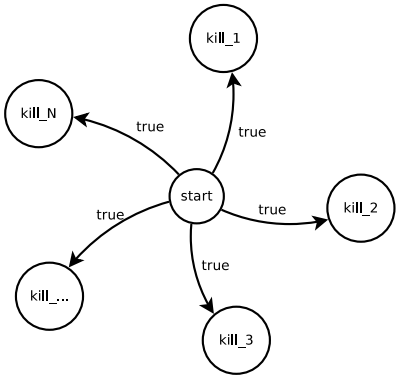

Fig. 14. The model of environment used for simulation

The robot was modelled as a two state machine which would transition from healthy to faulty state whenever it receives a kill message from the environment. The controller would be functional when the robot is in healthy state. The state machine is shown below

The state machines described in the previous section were implemented in MATLAB environment using Stateflow. MATLAB also provides a verification tool which was used for the verificaiton of properties of the swarm. The system was slightly modified before carrying out the verification. The modifications made involved setting the critical

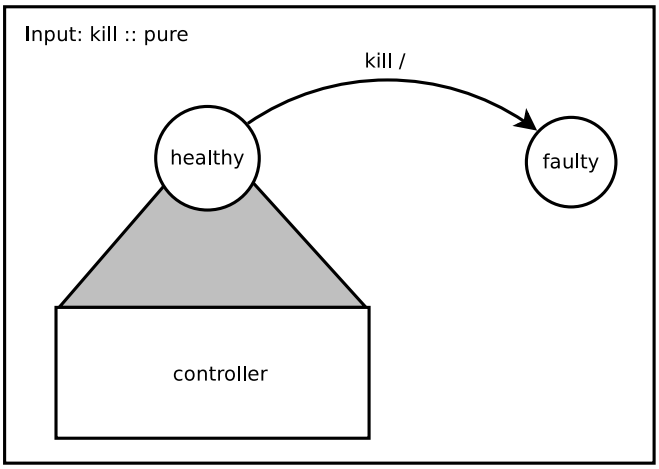

Fig. 15. Model of an agent in swarm

count for inferring failure to one. Then the tool was used to verify the safety property for this system.

The swarm always generates an `act` *message if a warning is generated*

VI. Lessons Learnt

This project required us to work in different domains of cyberphysical systems. We used the concepts learnt in the course for design and verification of the state machines. We also worked with communication hardware like XBee which required us to gain understanding of the concepts like peer-to-peer and broadcast communication. We also implemented a simple system which could use XBees (which have the ability to do P2P communication) for broadcasting information to a swarm. We developed the simulation using ROS and Gazebo and gained experience in using these systems.

VII. Future Scope

We have focussed only on a simple model for the proposed method due to the limited time frame. But this method has a lots of scope for improvement and extension. We enlist some of improvements which can be made in the proposed system.

1) The method assumes that there is no more than one fault at a time. Better neighborhoods can be defined which can be used to infer multiple faults.
2) The system assigns hard classes to robot as being faulty or non-faulty, a possible extension would be to use a probabilistic framework in which beliefs about robot health are updated over time. This can help in preventing faults and damage to robots by isolating them before a failure may occur.
3) Each robot is assigned a single *function* but it may be desirable, sometimes even necessary that multiple robots perform a single task, such cases can be handled by forming a hierarchy in which the nodes in this system are not individual robots but groups of robots themselves.
4) Better methods of deciding neighborhoods can be designed which take into account the range of communication.

References

[1] I. Navarro and F. Matía, "An Introduction to Swarm Robotics," *ISRN Robotics*, vol. 2013, pp. 1–10, 2013. [Online]. Available: http://www.hindawi.com/journals/isrn/2013/608164/

[2] M.-H. Lee and Y.-H. Choi, "Fault detection of wireless sensor networks," *Computer Communications*, vol. 31, no. 14, pp. 3469–3475, sep 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0140366408003587

APPENDIX: DEFINING A SWARM

A swarm is defined using a `launch` file in ROS. A sample swarm is shown below

```
<!--create a swarm of 4 active + 2 inactive robots-->
<launch>
    <!--active robots of the swarm-->
    <node pkg='swarm_node' type='node' name='node1' output='screen' >
        <param name='robot_id' value='1' />
        <param name='neighbors' value='[2,4]' />
        <param name='state' value='active' />
    </node>
    <node pkg='swarm_node' type='node' name='node2' output='screen' >
        <param name='robot_id' value='2' />
        <param name='neighbors' value='[1,3]' />
        <param name='state' value='active' />
    </node>
    <node pkg='swarm_node' type='node' name='node3' output='screen' >
        <param name='robot_id' value='3' />
        <param name='neighbors' value='[2,4]' />
        <param name='state' value='active' />
    </node>
    <node pkg='swarm_node' type='node' name='node4' output='screen' >
        <param name='robot_id' value='4' />
        <param name='neighbors' value='[1,3]' />
        <param name='state' value='active' />
    </node>

    <!--substitute (inactive) robots of the swarm-->
    <node pkg='swarm_node' type='node' name='node5' output='screen' >
        <param name='robot_id' value='5' />
        <param name='subs' value='[1,4]'/>
        <param name='state' value='inactive' />
    </node>
    <node pkg='swarm_node' type='node' name='node6' output='screen' >
        <param name='robot_id' value='6' />
        <param name='subs' value='[2,3]'/>
        <param name='state' value='inactive' />
    </node>
</launch>
```

Appendix: Stateflow design of the state machine