AFRL-AFOSR-UK-TR-2012-0011

Self-Stabilizing and Efficient Robust Uncertainty Management

Michael B. Segal

Ben Gurion University
Department of Communication Systems Engineering
Ben-Gurion Boulevard
Beer-Sheva, Israel 84751

October 2011

Final Report for 15 October 2008 to 15 October 2011

Air Force Research Laboratory
Air Force Office of Scientific Research
European Office of Aerospace Research and Development
Unit 4515 Box 14, APO AE 09421

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* 12 June 2012	**2. REPORT TYPE** Final Report	**3. DATES COVERED** *(From – To)* 15 October 2008 – 15 October 2011

4. TITLE AND SUBTITLE **Self-Stabilizing and Efficient Robust Uncertainty Management**	**5a. CONTRACT NUMBER** FA8655-09-1-3016
	5b. GRANT NUMBER **Grant 09-3016**
	5c. PROGRAM ELEMENT NUMBER 61102F
6. AUTHOR(S) Dr. Michael B. Segal	**5d. PROJECT NUMBER**
	5d. TASK NUMBER
	5e. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ben Gurion University Department of Communication Systems Engineering Ben-Gurion Boulevard Beer-Sheva, Israel 84751	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) EOARD Unit 4515 BOX 14 APO AE 09421	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/AFOSR/RSW (EOARD)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) **AFRL-AFOSR-UK-TR-2012-0011**

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited. (approval given by local Public Affairs Office)

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Ben Gurion University as follows: During the course of this research project we have examined various models of communication and UAVs coordination including simulating birds flocking behavior. We considered the situation when the members of the swarm may discover each other by exchanging messages; periodically collecting information concerning the position, speed and direction of other members. Next, we considered the situation where the means of communication are lost between the participants of the swarm. We have also presented information-theoretically secure schemes for sharing and modifying a secret among a dynamic swarm of computing devices. We studied the problem of topology control through power assignments so that the induced communication graph of UAVs is strongly connected under optimization objectives of energy efficiency, interference and stretch factor. We considered different models of aggregating information in mobile networks. We present the first set of swarm flocking algorithms that maintain connectivity while electing direction for flocking. We proposed a collaborative application monitoring infrastructure, that is capable of dramatically decreasing the susceptibility of mobile devices to malicious applications. We explored a polar representation of optical flow in which each element of the brightness motion field is represented by its magnitude and orientation instead of its Cartesian projections. Finally, we have proposed a fully automatic solver to reconstruct the complete image from a set of non-overlapping, unordered, square puzzle parts.

15. SUBJECT TERMS

EOARD, Control, Automated Reasoning

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18, NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** JAMES LAWTON Ph. D.
a. REPORT UNCLAS	**b. ABSTRACT** UNCLAS	**c. THIS PAGE** UNCLAS	SAR	50	**19b. TELEPHONE NUMBER** *(Include area code)* +44 (0)1895 616187

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39-18

Final report, October, 2011

During 3 year project on "Self-Stabilizing and Efficient Robust Uncertainty Management" our team produced more than a dozen of papers on this topic and also provided C++ code located at `http://www.cs.bgu.ac.il/~segal/flocks.html` and attached to this document. We have examined various models of communication and UAVs coordination including simulating birds flocking behavior. We considered the situation when the members of the swarm may discover each other by exchanging messages; periodically collecting information concerning the position, speed and direction of other members. Next, we considered the situation where the means of communication are lost between the participants of the swarm. We also presented information-theoretically secure schemes for sharing and modifying a secret among a dynamic swarm of computing devices. We studied the problem of topology control through power assignments so that the induced communication graph of UAVs is strongly connected under optimization objectives of energy efficiency, interference and stretch factor. We considered different models of aggregating information in mobile networks. We present the first set of swarm flocking algorithms that maintain connectivity while electing direction for flocking. We proposed a collaborative application monitoring infrastructure, that is capable of dramatically decreasing the susceptibility of mobile devices to malicious applications. We explored a polar representation of optical flow in which each element of the brightness motion field is represented by its magnitude and orientation instead of its Cartesian projections. We proposed a fully automatic solver to reconstruct the complete image from a set of non-overlapping, unordered, square puzzle parts.

Below we present our detailed finding regarding issues mentioned above and additional aspects of our research.

1. Swarm formation and swarm flocking may conflict each other. Without explicit communication, such conflicts may lead to undesired topological changes since there is no global signal that facilitates coordinated and safe switching from one behavior to the other. Moreover, without coordination signals multiple swarm members might simultaneously assume leadership, and their conflicting leading directions are likely to prevent successful flocking. To the best of our knowledge, we present the first set of swarm flocking algorithms that maintain connectivity while electing direction for flocking, under conditions of no communication in our joint paper "Direction Election in Flocking Swarms". The algorithms allow spontaneous direction requests and support direction changes. We develop simple and efficient algorithms for silent direction election, taking into account (bounded) environmental and parametric uncertainties, while providing a mechanism for connectivity preservation and collision avoidance. These latter capacities are obtained by introducing the notion of a *spring*, which resembles a potential function with restrictions and provides flexibility in the presence of uncertainties. The same mechanism also withstands temporary coexistence of multiple leaders. We do note that although direction election techniques do exist in the context of mobile robotics, these methods assume explicit communication between entities, which is outside the scope of our study.

2. In the paper "SPANDERS: Distributed Spanning Expanders", we consider self-stabilizing and self-organizing distributed construction of a spanner that forms an expander, which we term *spander*. We use folklore results to randomly define an expander graph. Given the randomized nature of our algorithms, a monitoring technique is presented for ensuring the desired results. The monitoring is based on the fact that expanders have a rapid mixing time and the possibility of examining the rapid mixing time by $O(n \log n)$ short ($O(\log^4 n)$ length) random walks even for non regular expanders. We then employ our results to construct a hierarchical sequence of spanders, each of them an expander spanning the previous one. Such a sequence of spanders may be used to achieve different quality of service assurances in different applications. Several snap-stabilizing algorithms that are used to utilize the monitoring are presented, including reset and token tracing algorithms for message passing systems.

   Distributed computing and communication networks research tries to define spanning sub-graphs, such as spanning (BFS, DFS) tree algorithms, or a spanner graph that preserves a stretch factor between the shortest path in the original graph and the spanning graph. Dynamic and fault-tolerant algorithms, such as self-stabilizing spanning trees, were extensively investigated. However, distributed fault-tolerant algorithms for defining and maintaining a prominent class of graphs, expander graphs, were not thoroughly explored. In this paper we consider edge expanders. A graph $G = (V, E)$ is an edge expander if there exists a constant c, such that for each set S of vertices (where $|S| < |V|/2$) it follows that $|E(S, \bar{S})|/|S| > c$. Expander graphs are perfect as a basis for communication networks; they are highly connected and symmetric in nature while being sparse and have a diameter in $O(\log n)$. Moreover, expander graphs are robust for dynamic changes; for example, removal of a constant number, k, of nodes from the network may disconnect only additional $O(k)$ nodes.

   Self-stabilizing and self-organizing distributed algorithms for distributed construction of spanders are presented. All of the algorithms assume a message passing system. We start by reviewing folklore results in randomized construction of expander graphs; first we consider the case in which the communication graph is a complete graph. In this simple case, a distributed random choice of a constant number of edges from each node results, with high probability, in an expander. Then we turn to consider the case of a communication graph that contains an expander, namely, has a certain expansion parameter. We show that when the spander edges are chosen using a binomial distribution, the obtained expansion is proportional to the expansion of the original graph, reducing the number of edges by the same factor.

3. The temporary physical topology of the network is determined by the distribution of the wireless stations as well as the transmission power of each station. The most fundamental problem in wireless ad-hoc networks is topology control via power assignments. In particular, there was an increasing interest in low cost spanner construction in wireless ad-hoc networks through topology control. In this paper we study asymmetric power assignments so that the induced communication graph

has a good distance and energy stretch simultaneously, with additional optimization objectives: both minimizing the total energy consumption, interference level, hop-diameter, and maximizing the network lifetime. In the paper "Improved Multi-criteria Spanners for Ad-Hoc Networks under energy and distance metrics", we consider two possible scenarios of node deployments random and deterministic. For n random nodes distributed uniformly and independently in a unit square we present several power assignments with varying construction time complexity. The probability of our results converges to one as the number of network nodes, n, increases. For the deterministic (arbitrary) layout of nodes we study the trade off between total energy consumption and distance stretch factor in the worst case, and then present two power assignments with non-trivial bounds. To the best of our knowledge, these are the first results for spanner construction in wireless ad-hoc networks which provide provable bounds simultaneously for both the energy and distance metrics. In addition, we explore the distance stretch factor of the minimum weight spanning tree, which is of independent interest in the spanner construction research.

One might think that applying traditional algorithms for the construction of spanners with low total edge weight in our model is possible. This is, however, not always possible for distance spanners, since the result of applying such an algorithm might result in a very large cost, due to our weight function. For energy spanners, it is possible to use the algorithms developed for general graphs, but it seems that better results can be achieved due to the fact that nodes are positioned in the plane and the weight function holds the weak triangle inequality. The generic algorithm consists of three steps as follows: (a) Select a subset of nodes $S \subset V, |S| = k$, which are relatively close to the remaining ones; (b) Compute k shortest path trees, rooted at the nodes from S. Assign powers to all nodes, so that these trees are subgraphs of the induced communication graph; (c) Increase the power of each node, if needed, to ensure that obtained construction is a subgraph of the induced communication graph.

4. In the paper "Deaf, Dumb, and Chatting Asynchronous Robots" we investigate avenues for the exchange of information (explicit communication) among deaf and dumb mobile robots scattered in the plane. We introduce the use of movement-signals (analogously to flight signals and bees waggle) as a mean to transfer messages, enabling the use of distributed algorithms among robots. We propose one-to-one deterministic movement protocols that implement explicit communication among asynchronous robots. We first show how the movements of robots can provide implicit acknowledgment in asynchronous systems. We use this result to design one-to-one communication among a pair of robots. Then, we propose two one-to-one communication protocols for any system of $n \geq 2$ robots. The former works for robots equipped with observable IDs that agree on a common direction (sense of direction). The latter enables one-to-one communication assuming robots devoid of any observable IDs or sense of direction. All three protocols (for either two or any number of robots) assume that no robot remains inactive forever. However, they cannot avoid that the robots move either away or closer of each others, by the way requiring robots with an infinite visibility.

In this paper, we also present how to overcome these two disadvantages. These protocols enable the use of distributing algorithms based on message exchanges among swarms of Stigmergic robots. They also allow robots to be equipped with the means of communication to tolerate faults in their communication devices.

All three protocols, either for two or n robots, are presented in the semi-synchronous model, imposing a certain amount of synchrony among the active robots, i.e., at each time instant, the robots which are activated, observe, compute, and move in one atomic step. However, no other assumption is made on the relative frequency of robot activations with respect to each other, except that each robot is activated infinitely often (uniform fair activation). This lack of synchronization among the robots prevents the robots either to move away of or to get closer to each other infinitely often. As a consequence, the robots are required to have an infinite visibility. Visibility capability of the robots is an important issue. In this paper, we also show how to overcome these two drawbacks by assuming that no robot can be activated more than $k \geq 1$ times between two consecutive activations of any other robot. Note that our protocols can be easily adapted to efficiently implement one-to-many or one-to-all explicit communication. Also, in the context of robots (explicitly) interacting by means of communication (e.g., wireless), since our protocols allow robots to explicitly communicate even if their communication devices are faulty, in a very real sense, our solution can serve as a communication backup, i.e., it provides fault-tolerance by allowing the robots to communicate without means of communication (wireless devices).

5. In the paper "Bounded-Hop Energy-Efficient Liveness of Flocking Swarms" we consider a set of n mobile wireless nodes, which have no information about each other. The only information a single node holds is its current location and future mobility plan. We develop a two-phase distributed self-stabilizing scheme for producing a bounded hop-diameter communication graph.

   The first phase is dedicated to the construction of an underlying topology for the dissemination of data needed for the second phase. In the second phase the required topology is constructed by means of an asymmetric power assignment under two modes — static and dynamic. The former aims to provide a steady topology for some time interval, while the latter uses the constant node locations changes to produce a constantly changing topology, which succeeds to preserve the required property of the bounded hop-diameter. More precisely, the **static mode**, preserves all the relevant communication links (those that are used for inducing the required topology) for the whole time interval $[t_s, t_f]$. Note that some other links might appear and disappear during the time interval, however the important links, which define the required topology remain unchanged. In other words, the communication graph, which is variant in time, always includes a subgraph which is unchanged for the whole time interval. The **dynamic mode** is different in that there is no constant subgraph which holds the topology property. However, as communication links are added and removed, depending on the movement of the nodes, the topology property requirement (e.g. connected dominating set) is satisfied during the entire period $[t_s, t_f]$.

We provide an $O(\lambda, \lambda^2)$-bicriteria approximation (in terms of total energy consumption and network lifetime, respectively) algorithm in the static mode: for an input parameter λ we construct a static h-bounded hop communication graph, where $h = n/\lambda + \log \lambda$. In the dynamic mode, given a parameter h we construct an optimal (in terms of network lifetime) h-bounded hop communication graph when every node moves with constant speed in a single direction along a straight line during each time interval. Our results were validated through extensive simulations.

6. New operating systems for mobile devices allow their users to download millions of applications created by various individual programmers, some of which may be malicious or flawed. In order to detect that an application is malicious, monitoring its operation in a real environment for a significant period of time is often required.

   Mobile devices have limited computation and power resources and thus are limited in their monitoring capabilities. In this paper we propose an efficient collaborative monitoring scheme that harnesses the collective resources of many mobile devices, "vaccinating" them against potentially unsafe applications.

   In the paper "Efficient Collaborative Application Monitoring Scheme for Mobile Networks", we suggest a new local information flooding algorithm called "TTL Probabilistic Propagation" (TPP). The algorithm periodically monitors one or more application and reports its conclusions to a small number of other mobile devices, who then propagate this information onwards. The algorithm is analyzed, and is shown to outperform existing state of the art information propagation algorithms, in terms of convergence time as well as network overhead. The maximal "load" of the algorithm (the fastest arrival rate of new suspicious applications, that can still guarantee complete monitoring), is analytically calculated and shown to be significantly superior compared to any non-collaborative approach.

   Finally, we show both analytically and experimentally using real world network data that implementing the proposed algorithm significantly reduces the number of infected mobile devices. In addition, we analytically prove that the algorithm is tolerant to several types of Byzantine attacks where some adversarial agents may generate false information, or abuse the algorithm in other ways.

7. A core abstraction for many distributed algorithms simulates shared memory; this abstraction allows to take algorithms designed for shared memory, and port them to asynchronous message-passing systems, even in the presence of failures. There has been significant work on creating such simulations, under various types of permanent failures, as well as on exploiting this abstraction in order to derive algorithms for message-passing systems.

   All these works, however, only consider permanent failures, neglecting to incorporate mechanisms for handling transient failures. Such failures may result from incorrect initialization of the system, or from temporary violations of the assumptions made by the system designer, for example the assumption that a corrupted message is always identified by an error detection code. The ability to automatically resume normal operation following transient failures, namely to be self-stabilizing, is an essential property that should be integrated into the design and implementation of systems.

Our paper "Sharing Memory in a Self-Stabilizing Manner" presents the first practically self-stabilizing simulation of shared memory that tolerates crashes. Specifically, we simulate a single-writer multi-reader (SWMR) atomic register in asynchronous message-passing systems where less than a majority of processors may crash. The simulation is based on reads and writes to a (majority) quorum in a system with a fully connected graph topology. A key component of the simulation is a new bounded labeling scheme that needs no initialization, as well as a method for using it when communication links and processes are started at an arbitrary state.

Our solution is to partition the execution of the simulation into epochs, namely periods during which the sequence numbers are supposed not to wrap around. Whenever a "corrupted" sequence number is discovered, a new epoch is started, overriding all previous epochs; this repeats until no more corrupted sequence numbers are hidden in the system, and the system stabilizes. Ideally, in this steady state, after the system stabilizes, it will remain in the same epoch (at least until all sequence numbers wrap around, which is unlikely to happen).

8. The paper "Energy Efficient Data Gathering in Multi-Hop Hierarchical Wireless Ad Hoc Networks" studies the problem of data gathering in hierarchical wireless ad hoc networks. In this scenario, a set of wireless devices generates messages which are addressed to the base station. As not all nodes can reach the base station through a direct transmission, messages are relayed by other devices in a multi-hop fashion. Data gathering without aggregation can cause a very high energy consumption in some of the nodes, namely nodes which are the direct descendants of the base station in the data gathering tree, since these nodes are responsible for relaying all the messages which are generated in the network and thus consume most of the energy. This phenomenon is especially severe in hierarchical networks where the nodes in the second layer have the highest number of transmissions, which results in a rapid depletion of the battery charges and network lifetime decrease. As wireless devices are usually deployed in hard-to-reach areas, battery replenishment is impractical or even impossible, which makes the issue of energy efficiency critical for successful network operation.

   We consider data gathering without aggregation, i.e. all the generated messages are required to reach the base station this is in contrast to the well studied problem of data gathering with aggregation, which appears to be significantly simpler. The above scheme may have poor performance in wireless networks with hierarchical architecture. The devices in the layer closest to the base station form a bottleneck in terms of energy consumption as they experience a very high volume of forward requests. Eventually, these nodes will be the first to run out of their battery charges which will cause connectivity losses. In this paper we focus on prolonging the network lifetime of hierarchical networks through efficient balancing of forward requests, which is NP-hard. We develop an linear programming based approximation scheme which produces a data gathering tree with network lifetime which is at most k times less than the optimal one, where k is the number of hierarchical layers. Our results are analytically proved and validated through simulations.

9. In the paper "Optimizing Performance of Ad-hoc Networks Under Energy and Scheduling Constraints" we study the problem of data gathering in multi-hop wireless ad hoc networks. In this scenario, a set of wireless devices constantly sample their surroundings and initiate report messages addressed to the base station. The messages are forwarded in a multi-hop fashion, where the wireless devices act both as senders and relays. We consider data gathering without aggregation, i.e. the nodes are required to forward *all* the messages initiated by other nodes (in addition to their own) to the base station. This is in contrast to the well studied problem of data gathering with aggregation, which is significantly simpler.

   As some nodes experience a larger load of forward requests, these nodes will have their battery charges depleted much faster than the other nodes – which can rapidly break the connectivity of the network. We focus on maximizing the network lifetime through efficient balancing of the consumed transmission energy. We show that the problem is NP-hard for two network types and develop various approximation schemes. Our results are validated through extensive simulations.

   We show a reduction from the restricted assignment case of ***Scheduling on Unrelated Parallel Machines*** (SoUPM) to the data gathering problem on 3-layered graphs with varying message size per node. In the restricted assignment case of SoUPM, we assign n jobs on m machines, where each job j has has a fixed assignment cost per machine, which can be either p_j or ∞ (i.e., it cannot be assigned). The goal is to minimize the total cost of the most congested machine (also known as the makespan). Given an instance of SoUPM, we map the jobs to bottom nodes with weight p_j and the machines to intermediate nodes with weight 0. Since both problems have the same optimization criteria, the reduction shows that the data gathering problem is NP-hard. To transform an instance of the data gathering problem to SoUPM, we create a job per bottom node (with weight equal to the number of messages that node needs to transmit), and a machine per intermediate node. For intermediate nodes, we also add a dedicated job with weight equal to the node's messages. To approximate the data gathering problem, we transform the input graph to SoUPM, and use Lenstra et al. 2-approximation algorithm for SoUPM.

10. We consider the problem of n agents wishing to perform a prescribed computation on common inputs in a secure manner. Security is defined by requiring that even if the entire memory contents of some of the agents is exposed, no information is revealed about the state of the computation. We place no a priori bound on the number of inputs. This problem has received ample attention recently in the context of swarm computing and Unmanned Aerial Vehicles (UAV) that collaborate in a common mission, as well as in the outsourcing of computation and storage to the cloud. Existing schemes achieve this notion of privacy for arbitrary computations, at the expense of one round of communication per input among the n agents. In the paper "Distributed Private Computation on Unbounded Global Inputs" we show how to avoid communication altogether during the course of

the computation on inputs of unbounded length, with the trade-off of computing a smaller class of functions, namely, those carried out by finite state automata. Our scheme, which is based on a novel combination of secret-sharing techniques and the Krohn-Rhodes decomposition of finite state automata, achieves the above goal in an information-theoretically secure manner, and, furthermore, does not require randomness during its execution.

We present a scheme that achieves the goal in an information-theoretically secure manner (i.e., there are no bounds imposed on the adverary's computational power), and does not require randomness during the execution. Our scheme is based on a novel combination of secret-sharing techniques and the Krohn-Rhodes decomposition of finite automata. Informally, Krohn-Rhodes theory states that any finite state automaton can be emulated by a combination (cascade productsee) of permutation automata and flip-flop automata. (A permutation automaton is an automaton such that each of its possible input symbols induces a permutation of the automatons states.) The computation complexity per each received input symbol, and the storage complexity required by our scheme are a function only of (the decomposition of) the automaton, and not of the number of symbols processed. A trade-off for this is that, depending on the automaton, the number of components of its Krohn-Rhodes decomposition might be exponential in its number of states. We note, however, that for many interesting and relevant automata, there is a small Krohn-Rhodes decomposition. We present an example of such an automata family with a small Krohn-Rhodes representation.

11. Energy efficiency is not the only challenge faced by the network designer. As nodes communicate through radio signals, wireless interference becomes inevitable. Simultaneous transmissions are sensed at every node, which may lead to incorrect signal receptions. We consider omnidirectional antennas, where the transmission of a single node is propagated in all directions. The level of interference depends on the transmitting nodes proximity and the transmission ranges. High levels of interference decrease the number of transmissions that can happen simultaneously, which has a direct affect on the schedule length of the network, which is the required number of time slots for the message to propagate from the source to all the other nodes in the network. It should be noted that traditional works which aim to minimize the hop-diameter in order to minimize the schedule length fail to do so as they neglect the presence of interference. We consider a fundamental topology control problem which is to induce an energy efficient broadcast communication backbone with a low schedule length. That is, given a special source node s (also referred to as the root node), we wish to induce a communication graph by adjusting transmission powers, so that there is a directed path from s to every other node in the network; the total energy consumption, network lifetime and feasible schedule length are used to measure the efficiency of the scheme.

    In the paper "Interference-Free Energy Efficient Scheduling in Wireless Ad Hoc Networks" we are interested in asymmetric power assignments so that the induced broadcast communication graph is both, energy efficient and has a short collision-free broadcast schedule. We consider both random and deterministic node layouts and develop four different broadcast schemes with provable performance guarantees on three optimization objectives simultaneously: total energy consumption, network lifetime and collision-free schedule length. We also show some numerical results which support our findings.

Interference is a direct consequence of any power assignment p. A signal transmitted over one communication link may interfere with the correct reception of a transmission over some other link. We adopt the **protocol interference model** which defines for each node u a set of nodes, $I_p(u,T)$, referred to as the *conflict set* of u, which consists of nodes which cannot be scheduled to transmit simultaneously with u because of interference to either the recipients of u or v, in a broadcast subtree T of H_p which is used for the broadcast task. That is, node u cannot be scheduled simultaneously with v iff there exists a child of u in T which is interfered by v or vice versa (a child of v interfered by u). Our algorithms are based on the following claim: If a power assignment p induces a broadcast arborescence H_p rooted at node $s \in V$ then there exists a feasible broadcast schedule S so that $Len(S) \leq h_s(H_p) \cdot (|I_p^*| + 1)$, where $Len(S)$ is the length of schedule and $h_s(H_p)$ is the length of longest path from s in H_p.

12. The paper "RoboCast: Asynchronous Communication in Robot Networks" introduces the RoboCast communication abstraction. The RoboCast allows a swarm of non oblivious, anonymous robots that are only endowed with visibility sensors and do not share a common coordinate system, to asynchronously exchange information. We propose a generic framework that covers a large class of asynchronous communication algorithms and show how our framework can be used to implement fundamental building blocks in robot networks such as gathering or stigmergy. In more details, we propose a RoboCast algorithm that allows robots to broadcast their local coordinate systems to each others. Our algorithm is further refined with a local collision avoidance scheme. Then, using the RoboCast primitive, we propose algorithms for deterministic asynchronous gathering and binary information exchange.

    In more details, we formally specify a robot network communication primitive, called RoboCast, and propose implementation variants for this primitive, that permit anonymous robots not agreeing on a common coordinate system, to exchange various information (e.g. their local coordinate axes, unity of measure, rendezvous points, or binary information) using only motion in a two dimensional space. Contrary to previous solutions, our protocols all perform in the fully asynchronous CORDA model, use constant memory and a bounded number of movements. Then, we use the RoboCast primitive to efficiently solve some fundamental open problems in robot networks. We present a fully asynchronous deterministic gathering and a fully asynchronous stimergic communication scheme.

    Our algorithms differ from previous works by several key features: they are totally asynchronous (in particular they do not rely on the atomicity of cycles executed by robots), they make no assumption on a common chirality or knowledge of the initial positions of robots, and finally, each algorithm uses only a bounded number of movements. Also, for the first time in these settings, our protocols use CORDA-compliant collision avoidance schemes.

13. Most autonomous agents, UAVs in particular, observe the environment under relative motion (i.e., either the agent and/or parts of its environment move). This fact creates a critical need for robust motion estimation for various tasks, from egomotion estimation to target detection and tracking. One elementary construct in this field is the so called "optic flow", the dense vector field which describes the motion of luminance patterns in the agent's field of view. While the literature on optic flow estimation is vast, the solutions proposed in the last three decades have much in common, and in particular, they are all based on a similar representation of the optic flow in a Cartesian frame of reference. As was argued recently, at least in some important cases (such as when the scene contains specular objects or fluid flows), these solutions are not satisfactory. To explore an important alternative, here we explore a polar representation of optical flow in which each element of the brightness motion field is represented by its magnitude and orientation. This seemingly small change in representation provides more direct access to the intrinsic structure of a flow field, and when used with existing variational inference procedures it provides a framework in which regularizers can be intuitively tailored for very different classes of motion. Indeed, in the paper "A Polar Representation of Motion and Implications for Optical Flow" we develop and derive the necessary mathematics for expressing the common optical flow constraints (including the brightness constancy and piecewise continuity) in polar coordinates and examine the implications. Expressing the problem this way also creates several complications, such as the need to cope with the periodicity of the flow orientation or the fact that its magnitude cannot be negative. We suggest practical solutions and approximations and we examine the performance of the derived algorithms on several classes of motions.

    The evaluations of the polar representation for optical flow first reveals that using this representation makes certain statistical properties of optical flow more explicit and accessible. For example, it reveals greater independence between the flow components (orientation and magnitude) than in Cartesian representation, and it shows qualitatively different behaviors depending on the type of optic flow at hand. In particular, one can show that fluid flows (e.g., due to turbulence in the air) are qualitatively different than specular flows or rigid body flows when examined in the polar representation. This itself is a prior knowledge that can be leveraged in the estimation of optical flow from images, and it is *not* acceptable in Cartesian representation. When it comes to quantitative performance of optic flows, we show that algorithms based on the polar representation can perform as well or better than the state-of-the-art when applied to traditional optical flow problems concerning camera or rigid scene motion, and at the same time, it facilitates both qualitative and quantitative improvements for non-traditional cases such as fluid flows and specular flows, whose structure is very different. This suggests that all agents, UAVs in particular, should employ novel optic flow algorithms based on this representation,when the accuracy of estimation is a critical factor.

14. One problem of distributed sensing that is highly relevant for swarms of UAVs can be described as follows: given a set of *local* sensory snapshots (say, small images) of the "world", each acquired by

one agent while being ignorant of its geo-spatial configuration (e.g., due to lack of proper sensors or wartime GPS blackout), is it possible to create a global description of the world by properly assembling all local evidences into one coherent image. In the abstract, this problem is like a puzzle solving challenge - given many pictorial pieces, is it possible to assemble them into the picture from which they were cut. Unfortunately, one cannot check all possible combinations and evaluate each for correctness. This is simply non tractable and the problem is indeed known to be NP-Complete. Even worse, unlike in the famous jigsaw game, here we can assume no knowledge of the target image, and thus the evaluation stage is ill defined also. The only piece of evidence one can leverage is that the assembled puzzle should be visually coherent, a constraint that must be formalized in order to become constructive. In this work we have studies this problem with one additional difficulty, where the pieces all have the same (Square) shape and thus selecting the order of pieces and the evaluation of correctness must rely on appearance only and not on the shape of adjacent pieces. We should mention that the state of the art up to this work were algorithms that could process puzzles of up to 432 pieces. Depending on how one measures performance, these past solutions provided assembled puzzles accurate on average no more than 50%, and hardly ever could solve a puzzle completely. Furthermore, these solutions needed some "hint", for example by providing the exact location of several pieces (a.k.a. anchors) or a low resolution version of the assembled puzzle.

To handle the puzzle problem more effectively (and in effect to raise the state-of-the-art many folds), we, in our paper "A fully automated greedy square jigsaw puzzle solver", made several contributions. We introduce an iterative greedy solver which combines both informed piece placement and rearrangement of puzzle segments to find the final solution. Among our other contributions are new compatibility metrics which better predict the chances of two given parts to be neighbors, and a novel estimation measure which evaluates the quality of puzzle solutions without the need for ground-truth information.

Results:

Incorporating the contributions above, our new approach for puzzle solving facilitates solutions that surpass state-of-the-art solvers on puzzles of size larger than ever attempted before. On puzzles of 432 pieces (thus far the state-of-the-art in size) our accuracy performance stepped up to approximately 95% on average (improvement from 10%-55%, depending on how one measures), with 65% cases solved 100% accurately. Our solver is able to cope equally well with problems consisting close to 1000 pieces, and success has been demonstrated on puzzles with thousands of pieces also (although no statistics has been obtained for those). The figure below shows one such example for demonstration. Please refer to the paper and the project web site (`https://sites.google.com/site/greedyjigsawsolver/`) for additional examples.

We certify that there were no subject inventions to declare during the performance of this grant.

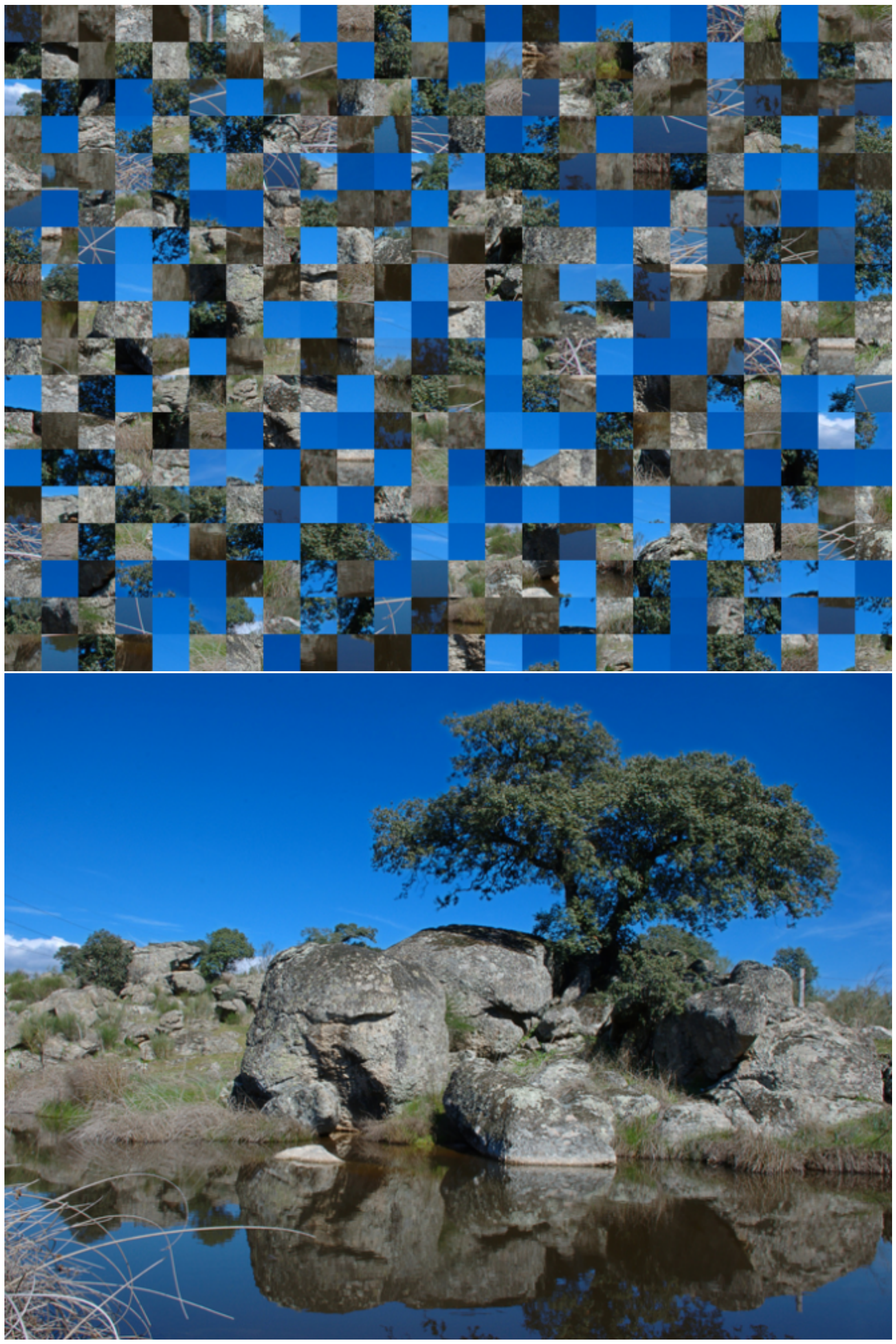

Figure 1: A sampled solution of our proposed algorithms for the creation of a coherent global image from the collection of distributed local snapshots. On the top is the collection of local image pieces. On the bottom in the assembled global image.

Final Project Code

```
/***********************************************************************
***********
 *  Self-Stabilizing and Efficient Robust Uncertainty Management
 *
************************************************************************
**********/



/***********************************************************************
***********
 *  Class name: uav
 *  Description:  This class define an object which represent the UAV
with all
 *                the information that the UAV has about itself and
it's other
 *                UAVs. This class is where all calculations are
taken place and
 *                all information arrays and variables are saved.

************************************************************************
**********/


#include <queue.h>
#include <omnetpp.h>
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <sstream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <coutvector.h>
#include <chistogram.h>

#include "uavm_m.h"
#include "uavInformation.h"
#include "uavDist.h"

namespace uav {


class uav : public cSimpleModule
{

/***********************************************************************
**********
 * Here we define all functions in which we will use during the
simulation

************************************************************************
*********/
  public:
     uav();
     virtual ~uav();
     void randomTime();//this function generates a random time and
wait it before calculations
     void findNeigbors();//this function check for every other uav if
it is within the R distance from this
     double* alignmentCalc(); //this function finds the average
flight direction of all visible uavs
     double* cohisionCalc();//this function finds the center of mass
position wise
     void avoidColision();// if there is a uav in C distance of the
uav - avoids collision
     void newPosition();// this function finds the new position for
the uav
     void initialPosition();//this function insert the initial
position from the display string
     void isNeighborCalc(uavm *msg);//this function check if the uav
is within a certine uav's R
     void updateUavInfo(uavm * msg);//this function updates the uav
information array
     void separation(int Id);//this function separate the uav from a
near uav obstacle
     void createSpanner();//calculate the spanner
     void sendData();//send out a large size message and calculate
the battry level
     void ignoringRandomNei(int isParticipate[]);//ignoring a random
number of neighbors
     void addingDev(double* DevSum);//adding delta to each neighbor's
direction.
     void addingDevForMe(double* DevSum);//adding delta to my
direction.
     void startLead();//starting leading interval
     void saveConnectivity();//calculating the leader fly direction
     void finish();//this function is called when the simulation ends
and collect information

  protected:
    virtual void initialize();//this function is called at the very
begining of the simulation and
                              // it's responsible of initializing all
needed variables and arrays.
    virtual void handleMessage(cMessage *msg);//This function is
called when ever the uav receive a message.
    virtual uavm *generateMessage(char *name);//This function create
a message of type uavm message which contain the UAV's
                                              //id,position and
direction and return this message.
    virtual uavm *generateMessage(char *name,int dest);//This
function create a message of type uavm message which contain the
UAV's
                                                         //id,position
and direction and return this message.


/**********************************************************************
**********
 * Here we define all variables in which we will use during the
simulation
```

```
**********************************************************************
*********/
  public:



     int R;//the detection and communication radius
     int C;//the separation radius
     int velocity;//the uav's speed
     int cohVelFactor;// the uav side-ways speed =
velocity/CohVelFactor
     int Rneighbors;// the neighbor search radius
     int maxUavs;//the total number of uavs that left the base
     int position[3];//the current position of the uav [x],[y],[z] =
a point in space
     int finalGoal[3];//the uav's target
     int ID;//The uav's id
     int dataSent;//the number of packets of data sent
     int neighborNum;//the number of current neighbors
     int RSpanner;//this is the optimum radius for data trans found
by algorithm
     int turn;//indicates the current leader.
     double battery;//This simulate the battery power
     double diraction[3];//the current direction of the uav
[x],[y],[z] = a point in space
     double P_D_ratio;//the power to distance unit ratio
     bool batterySaveMode;//indicate if the spanner option is on or
off
     bool dataFirst;//indicate if this is the first time data is
being sent
     bool firstTime;//indicate if this is the first time a spanner is
built
     bool err;//indicate an error of send
     bool tooBig;//indicate a radius outside the boundries of R
     bool imLead;//while it's the UAV turn = true
     bool firstLead;//the first time that a leader is selected

     cDisplayString dispStr;//the display string of the uav
     uavInformation *uavInfo;//the array of objects that hold the
UAVs information

     cOutVector cVect;//this vector record the battery level change
in time
     cOutVector cVect2;//this vector record the data packets send
     cOutVector cVect3;//this vector record the simulation ending
time
     cOutVector cVect_all_x;
     cOutVector cVect_all_y;
     cOutVector cVect_all_z;
     cOutVector cVect_lead_x;
     cOutVector cVect_lead_y;
     cOutVector cVect_lead_z;
     cLongHistogram cHist;
     cLongHistogram cHist_all_x;
     cLongHistogram cHist_all_y;
     cLongHistogram cHist_all_z;
     cLongHistogram cHist_lead_x;
     cLongHistogram cHist_lead_y;
     cLongHistogram cHist_lead_z;
```

```
/*********************************************************************
**********
 * Here we define all messages types in which we will use during the
simulation

*********************************************************************
*********/
     cMessage *interval;//This message is sent in every calculation
interval
     cMessage *initial;//This message is sent after the initialize
function is
                       //called and start the first sequence
     cMessage *rt;//This message is a random time message
     cMessage *neighborCheck;//This message is sent in order to find
neighbors, this
                             //message contain information about the
uav's position
                             //direction ect
     cMessage *neighborReply;//This message is a replay message to
the neighborCheck
                             //message which acknowledge that the uav
that sent this
                             //message is indeed a neighbor. this
message contain
                             //information about the neighbor's
direction and position
     cMessage *startCalc;//This message indicate the start of the
calculation of the
                         //new position for the uav
     cMessage *spannerInterval;//This message indicate the start of
the spanner build
                               //interval
     cMessage *data;//This message is the heavy duty data which is
being exchanged

     cMessage *leaderInterval;//This message starts the leading
section

     cMessage *leaderMove;//This message starts the leading direction
calc

     cMessage *myTurn;//This message changes the turn of whos leading

     cMessage *stopLeading;//This message stops the current leading
period





};

Define_Module(uav);

/*********************************************************************
****************
 * Function name: uav
 * Input: none
 * Output: void
 *
 * Description: This function is the object constructor and we keep
it on default

*********************************************************************
**************/
uav::uav()
{

}

/********************************************************************
****************
 * Function name: ~uav
 * Input: none
 * Output: void
 *
 * Description: This function is the object destructor and on which
we delete arrays
 *               and free their memory

*********************************************************************
**************/
uav::~uav()
{
      delete [] diraction;
      delete [] position;
      delete [] finalGoal;
      delete  uavInfo;
}
/********************************************************************
****************
 * Function name: initialize
 * Input: none
 * Output: void
 *
 * Description: this function is called at the very begining of the
simulation and
 *               it's responsible of initializing all needed variables
and arrays.
 *               In this function we also initialize all parameters
needed from the
 *               omnetpp.ini file.

*********************************************************************
**************/
void uav::initialize()
{

// initializing variables from the omnetpp.ini file
      R = par("R");
      C = par("C");
      batterySaveMode=par("batterySaveMode");
      maxUavs = par("maxUavs");
      velocity = par("velocity");
      battery = par("battery");
      dataSent=0;
      cVect.setName("battery");
      cVect_all_x.setName("Swarm Position[x]");
      cVect_all_y.setName("Swarm Position[y]");
      cVect_all_z.setName("Swarm Position[z]");
      cVect_lead_x.setName("Leader Position[x]");
      cVect_lead_y.setName("Leader Position[y]");
      cVect_lead_z.setName("Leader Position[z]");
      P_D_ratio=0.001;
      dataFirst=true;
      firstTime=true;
      ID = this->getId()-2; // assign an ID to each uav
      uavInfo=new uavInformation[maxUavs];
      turn = 0;
      imLead = false;
      firstLead = false;

// The uav start from a random position in the battlefield
// generates parameters from 0 to 6000
      diraction[0]= rand()%6000;
      diraction[1]= rand()%6000;
      diraction[2]= rand()%6000;

      position[0]=0;
      position[1]=0;
      position[2]=0;
      cohVelFactor = 8;


      WATCH(maxUavs);
      WATCH(position[0]);
      WATCH(position[1]);
      WATCH(position[2]);
      WATCH(ID);
      WATCH(R);
      WATCH(C);
      WATCH(batterySaveMode);
      WATCH(diraction[0]);
      WATCH(diraction[1]);
      WATCH(diraction[2]);

// In order to get the simulation going we schedule a self message
called initial
      initial = new cMessage("initial");
      scheduleAt(simTime(), initial);
}


/*********************************************************************
****************
 * Function name: handleMessage
 * Input: cMessage *msg
 * Output: void
 *
 * Description: This function is a build in function of the omnet
simulation.
 *                This function is called when ever the uav receive a
message.
 *                The messages can be either a self messages that was
scheduled by the
 *                uav program to arrive to self in a specific time or a
message
 *                received by another uav.
 *                According to each message type (it's name) the
function process the
 *                message and define appropriate action that needs to
be executed on
```

```
 *               receiving the message.

**********************************************************************
**************/
void uav::handleMessage(cMessage *msg)
{
//This if distinguish between a self pre scheduled message and
outside received messages

      if (msg->isSelfMessage())//The self messages types are:
      {
            //This message is sent in every calculation interval
            if(msg->isName("interval"))
            {
                  //Positioning the uav in the battlefield according
to the new position calculated
                  //in the last interval.
                  dispStr = getDisplayString();
                  dispStr.setTagArg("p",0,position[0]);
                  dispStr.setTagArg("p",1,position[1]);
                  if(batterySaveMode && !dataFirst)
                  {
                        if(RSpanner>R)
                        {
                              dispStr.setTagArg("r",0,R);
                        }
                        else
                        {
                              dispStr.setTagArg("r",0,RSpanner+200);
                        }
                  }

                  setDisplayString(dispStr);
                  neighborNum=0;//every interval we find the current
neighbors so we give 0
                  randomTime();//According to the symmetry breaking
rules each interval we
                               //produce a random time that we wait
before calculating
                               //so here in the beginning of the
interval we call the function
                               //that randomize this time and
schedule the rt message
            }

            //This message is a random time message produced and
schedule by randomTime function
            if(msg->isName("randomTime"))
            {
                  //the first thing we do after waiting a random time
is to find the neighbors
                  findNeigbors();
            }

            //This message is sent after the initialize function is
called and start
            //the first sequence.
            if(msg->isName("initial"))
            {
                  //we call the initialPosition function which place
the uav in his random
```

```
                        //initial position.
                        initialPosition();
                }

                //This message indicate the start of the calculation of
the
                //new position for the uav
                if(msg->isName("startCalc"))
                {
                        EV<<"I'm Lead is: "<<imLead<<" MY ID is
:"<<ID<<endl;
                        //we call the newPosition function which calculate
the new position of the
                        //uav in this interval.
                        if(!imLead)
                                newPosition();
                }

                //This message indicate the start of the spanner build
                //interval
                if(msg->isName("spannerInterval"))
                {
                        //we call the createSpanner function which
calculate the new spanner
                        //in this spanner interval and assign the
transmission range.
                        createSpanner();
                }

                //This message is the heavy duty data which is being
exchanged
                if(msg->isName("data"))
                {
                        //we call the sendData function which sends the
heavy duty data.
                        sendData();
                }
                if(msg->isName("leaderInterval"))
                {


                        myTurn = new cMessage("myTurn");
                        imLead = false;
                        //turn+=1;
                        stopLeading = new cMessage("stopLeading");
                        scheduleAt(simTime()+500, stopLeading);
                        startLead();
                        if(ID+1<maxUavs)
                                send(myTurn, "g$o", ID+1);

                }
                if(msg->isName("stopLeading")){
                        imLead=false;
                        if(ID==maxUavs-1)
                                endSimulation();

                }
                if(msg->isName("leaderMove"))
                {
                        if(imLead)
                        {
```

```
                        saveConnectivity();

                  }
                  else
                  {
                        startCalc = new cMessage("startCalc");
                        scheduleAt(simTime(), startCalc);
                  }
            }
      }

      else //The outside received messages types are:
      {
            //This message is sent in order to find neighbors, this
            //message contain information about the uav's position
          //direction ect
            if(msg->isName("neighborCheck"))
            {
                  //we cast the message received in order to get all
the information from the message
                  uavm *neighborCheck = check_and_cast<uavm *>(msg);
                  //we call updateUavInfo function which keeps the
required information on the uav
                  updateUavInfo(neighborCheck);
                  //we call isNeighborCalc function which determine
whether the uav is my neighbor
                  isNeighborCalc(neighborCheck);

                  //we check if the new neighbor is too close
(distance lower than c)
                  if(C>=uavInfo[neighborCheck-
>getSource()].currentDist)
                  {
                        //if the new neighbor is indeed too close we
deploy the separation
                        //function which avoid colliding with the
uav.
                        EV<<"AVOID COLLISTION"<<endl;
                        separation(neighborCheck->getSource());
                        if(position[0]==uavInfo[neighborCheck-
>getSource()].pos[0])
                        {
                              if(position[1]==uavInfo[neighborCheck-
>getSource()].pos[1])
                              {

      if(position[2]==uavInfo[neighborCheck->getSource()].pos[2])
                                    {
                                          EV<<"me : "<<ID<<" COLLIDE
with : "<<neighborCheck->getSource()<<endl;
                                    }
                              }
                        }
                  }
            }

            //This message is a replay message to the neighborCheck
            //message which acknowledge that the uav that sent this
            //message is indeed a neighbor. this message contain
            //information about the neighbor's direction and
position.
```

```
            if(msg->isName("neighborReply"))
            {
                  //we cast the message received in order to get all
the information from the message
                  uavm *neighborReply = check_and_cast<uavm *>(msg);
                  //we call updateUavInfo function which keeps the
required information on the uav
                  updateUavInfo(neighborReply);

                  //we check if the new neighbor is too close
(distance lower than c)

                  if(C>=uavInfo[neighborReply-
>getSource()].currentDist)
                  {
                        //if the new neighbor is indeed too close we
deploy the separation
                        //function which avoid colliding with the
uav.
                        EV<<"AVOID COLLISTION"<<endl;
                        separation(neighborReply->getSource());
                        if(position[0]==uavInfo[neighborReply-
>getSource()].pos[0])
                        {
                              if(position[1]==uavInfo[neighborReply-
>getSource()].pos[1])
                              {

      if(position[2]==uavInfo[neighborReply->getSource()].pos[2])
                                    {
                                          EV<<"me : "<<ID<<" COLLIDE
with : "<<neighborReply->getSource()<<endl;
                                    }
                              }
                        }

                  }
            }
            if(msg->isName("myTurn")){
                  leaderInterval = new cMessage("leaderInterval");
                        scheduleAt(simTime()+500,
leaderInterval);
            }
      }
      //after dealing with the message we delete it
      delete msg;
}

/*********************************************************************
****************
 * Function name: updateUavInfo
 * Input: uavm *msg
 * Output: void
 *
 * Description: This function is called when ever a message from a
near by uav is
 *             received. The message from the near by uav (*msg) is
a special message
 *             that was constructed by us and in this message there
is information
 *             about the uav's id' current position, direction ect.
 *              In this function we receive the message with this
information
 *              and insert it into an array that contain a structure
designed to hold
 *              this information.

*********************************************************************
**************/
void uav::updateUavInfo(uavm *msg)
{
      int Id;
      //extracting the uav's id from the message
      Id = msg->getSource();
      uavInfo[Id].participate = true;
      //extracting the uav's position and direction from the message
and entering it
      //in the right place in the array.
      uavInfo[Id].pos[0]= msg->getPosX();
      uavInfo[Id].pos[1]= msg->getPosY();
      uavInfo[Id].pos[2]= msg->getPosZ();
      uavInfo[Id].dir[0]= msg->getDirX();
      uavInfo[Id].dir[1]= msg->getDirY();
      uavInfo[Id].dir[2]= msg->getDirZ();

      //Calculating the distance between the nearby uav and us
      double distance;
      distance = sqrt(pow(position[0]-msg-
>getPosX(),2)+pow(position[1]-msg->getPosY(),2)+pow(position[2]-msg-
>getPosZ(),2));
      //entering the distance in the right place in the array
      uavInfo[msg->getSource()].currentDist=distance;

      //Calculating the distance between the nearby uav and us in all
dimensions
      uavInfo[Id].fromHimtoMe[0] =  (double)position[0] - msg-
>getPosX();
      uavInfo[Id].fromHimtoMe[1] =  (double)position[1] - msg-
>getPosY();
      uavInfo[Id].fromHimtoMe[2] =  (double)position[2] - msg-
>getPosZ();

      if(position[0]==uavInfo[msg->getSource()].pos[0])
      {
            if(position[1]==uavInfo[msg->getSource()].pos[1])
            {
                  if(position[2]==uavInfo[msg->getSource()].pos[2])
                  {
                        EV<<"me : "<<ID<<" COLLIDE with : "<<msg-
>getSource()<<endl;
                  }
            }
      }

}

/*********************************************************************
***************
 * Function name: isNeighborCalc
 * Input: uavm *msg
 * Output: void
 *
 * Description: This function is called when ever we receive a
neighborCheck message
 *              and we like to check whether this uav from which we
received this
 *              message is indeed a neighbor (or in other words: is
the distance
 *              between me and this uav is smaller than the detection
range R)

**********************************************************************
**************/
void uav::isNeighborCalc(uavm *msg)
{
      double distance;
      //Calculating the distance between the nearby uav and us
      distance = sqrt(pow(position[0]-msg-
>getPosX(),2)+pow(position[1]-msg->getPosY(),2)+pow(position[2]-msg-
>getPosZ(),2));
      //if this distance is under R (the detection range) than we add
this uav as
      //our neighbor and we create and send a neighborReply message
to that uav.

      if (R >= distance)
      {
            neighborNum++;
            //generating a special message containing confirmation
that now the
            //uav is my neighbor and my required information.
            neighborReply =
generateMessage((char*)("neighborReply"),msg->getSource());
            EV<<"Sending to: "<<msg->getSource()<<endl;
            send(neighborReply, "g$o", msg->getSource());

      }
}

/*********************************************************************
****************
 * Function name: createSpanner
 * Input: none
 * Output: void
 *
 * Description: This function create an h bounded spanner tree
according to
 *              Prof. Micheal Segal's article.
 *              Once all of the uavs are in a group mode (all of the
uavs are within
 *              detection range from me) this function is called and
using the
 *              information from the neighbors uavs a spanner is
created.
 *              The way the spanner is created is explained  in
detail manner in
 *              the final project report.
 *              In the end of this function a transmission range is
assigned
 *              to the uav and the time for reconstructing the
spanner is set.
```

```
**********************************************************************
**************/
void uav::createSpanner()
{
      queue<uavDist> buffDist;//This is a queue that hold objects of
type uavDist
                              //this queue will be used in order to
keep the order of
                              //which nodes will be examined and
assigned in the
                              //process of creating the spanner.
      double nodeAssigment[maxUavs];//This array will eventually hold
all of the
                                    //transmission ranges assignments
      uavDist allDist[maxUavs][maxUavs];//A matrix that holds all of
the distances from
                                        //all uavs to everyone.
      uavDist *p,*minDist,*temp,*temp2;
      bool stop,first;
      int i,j,k,minIndex,maxMinDist,brk;
      double dis;
      stop=true;
      first=true;
      err=false;
      tooBig=false;

      //initializing all nodes transmission ranges assignments
      for(i=0;i<maxUavs;i++)
      {
            nodeAssigment[i]=0;
      }

      //Calculating all possible distances from all uavs to all uavs
      for(i=0;i<maxUavs;i++)
      {
            for(j=0;j<maxUavs;j++)
            {
                  if(i==j)
                  {
                        p=new uavDist(0,i,i);
                        allDist[i][j]=*p;
                  }
                  else
                  {


                        if(i==ID)
                        {
                              dis=sqrt(pow(position[0]-
uavInfo[j].pos[0],2)
                                          +pow(position[1]-
uavInfo[j].pos[1],2)
                                          +pow(position[2]-
uavInfo[j].pos[2],2));


                              p=new uavDist(dis,j,i);
                              allDist[i][j]= *p;

                        }
                        else
                        {
                              if(j==ID)
                              {
                                    dis=sqrt(pow(uavInfo[i].pos[0]-
position[0],2)

      +pow(uavInfo[i].pos[1]-position[1],2)

      +pow(uavInfo[i].pos[2]-position[2],2));

                                    p=new uavDist(dis,j,i);
                                    allDist[i][j]= *p;
                              }
                              else
                              {
                                    dis=sqrt(pow(uavInfo[i].pos[0]-
uavInfo[j].pos[0],2)

      +pow(uavInfo[i].pos[1]-uavInfo[j].pos[1],2)

      +pow(uavInfo[i].pos[2]-uavInfo[j].pos[2],2));


                                    p=new uavDist(dis,j,i);
                                    allDist[i][j]= *p;
                              }

                        }
                  }
            }
      }
    //here we sort the matrix of all possible distances from low to
high
      for(i=0;i<maxUavs;i++)
      {
            for(j=0;j<maxUavs;j++)
            {
                  minDist = new uavDist(allDist[i][j].dist,
allDist[i][j].to,allDist[i][j].from);
                  minIndex=j;
                  for(k=j;k<maxUavs;k++)
                  {
                        if(allDist[i][k].dist < minDist->dist)
                        {
                                    minDist=new
uavDist(allDist[i][k].dist, allDist[i][k].to, allDist[i][j].from);
                                    minIndex=k;
                        }
                  }
                  temp=new
uavDist(allDist[i][j].dist,allDist[i][j].to,allDist[i][j].from);
                  allDist[i][j]=*minDist;
                  allDist[i][minIndex]=*temp;
            }
      }

    // find the maximum minimum distance that has not been assigned
yet
      maxMinDist=0;
      minIndex=0;
```

```
	for (i=0;i<maxUavs;i++)
	{
		if(nodeAssigment[i]==0 && allDist[i][1].dist>maxMinDist)
		{
			maxMinDist=allDist[i][1].dist;
			minIndex=i;
		}
	}

	//after finding the desired distance and the two nodes 
respectively we create
	//new objects with their information and add them to the queue.
	temp=new 
uavDist(allDist[minIndex][1].dist,allDist[minIndex][1].to,allDist[min
Index][1].from);
	temp2=new 
uavDist(allDist[minIndex][1].dist,allDist[minIndex][1].from,allDist[m
inIndex][1].to);
	buffDist.push(*temp);
	buffDist.push(*temp2);
	//We assign those two nodes which are needed for connectivity 
their assignments
	nodeAssigment[temp->from]=temp->dist;
	nodeAssigment[temp2->from]=temp->dist;
	brk=0;

	//Now after establishing the first nodes that from them we 
shell start to build
	//the spanning tree. This do-while loop will keep adding new 
nodes to be added
	//to the tree and assigning transmission ranges. This loop is 
ended once all nodes
	//are assigned and the queue is empty.
	do
	{
		//Loaded with the first two nodes we keep popping nodes 
from the queue and
		//checking whether those node's neighbors are found 
within the current assignment
		//and if so they are pushed into the queue. When the 
queue is empty we check
		//all nodes that were not added in this round to the 
spanner and again add
		//the node with the minimum distance to the elements of 
the spanning tree.
		while(!buffDist.empty())
		{
			//poping and saving the first node from the queue
			*temp=(uavDist)buffDist.front();
			buffDist.pop();
			i=1;
			//Checking what nodes fall inside of the 
broadcasting range of the
			//already assigned node and placing them in queue 
in order to assign
			//them later.
			while(i<maxUavs && allDist[temp-
>from][i].dist<temp->dist && temp->from!=i)
			{
				buffDist.push(allDist[temp->from][i]);
				i++;
```

```
                  }

                  //assigning the participant nodes with the
correspondent ranges according
                  //to our calculations.
                  if(nodeAssigment[temp->to]==0)
                  {
                        nodeAssigment[temp->to]=temp->dist;
                        if(!first)
                        {
                              nodeAssigment[temp->from]=temp->dist;
                        }
                  }
            }

            //Checking whether all nodes were assigned
            stop=true;
            for(i=0;i<maxUavs;i++)
            {
                  if (nodeAssigment[i]==0)
                  {
                        stop=false;
                  }
            }

        // find the max minimum distance that has not been assigned
yet and connect
            //it to the graph by pushing the two nodes into the
queue.
            bool z;
            z=false;
            maxMinDist=9999;
            minIndex=0;
            for (i=0; i<maxUavs && !stop ;i++)
            {
                  if(nodeAssigment[i]==0)
                  {
                        for(j=1; j<maxUavs; j++)
                        {
                                    if(nodeAssigment[j]!=0 &&
allDist[i][j].dist<maxMinDist )
                                    {
                                          z=true;
                                          maxMinDist =
allDist[i][j].dist;
                                          temp = new
uavDist(allDist[i][j].dist,allDist[i][j].to,allDist[i][j].from);
                                          temp2 = new
uavDist(allDist[i][j].dist,allDist[i][j].from,allDist[i][j].to);
                                    }
                        }

                  }
            }
            if(z)
            {
                  buffDist.push(*temp);
                  buffDist.push(*temp2);
            }
        brk++;
        if(brk==1000 && !stop)
        {
                stop=true;
                for(i=0;i<maxUavs;i++)
                {
                        if (nodeAssigment[i]==0)
                        {
                                nodeAssigment[i]=9999;
                        }
                }
        }
        //it is needed to know whether this is the first time we
assigned or not
        if(first)
        {
                first=false;
        }
        }while(!stop);//end of the do-while loop. this loop stops when
all nodes are
                    //assigned with transmission ranges.

        //We check that no mistake was made during the assignment
process.
        for(i=0;i<maxUavs;i++)
        {
                if (nodeAssigment[i]==9999)
                {
                        err=true;
                }
                if (nodeAssigment[i]!=9999 && nodeAssigment[i]>R)
                {
                        tooBig=true;
                }
        }

        //Assigning the uav with his own transmission range
        RSpanner=nodeAssigment[ID];
        //Scheduling the next spanner building interval
        spannerInterval = new cMessage("spannerInterval");
        scheduleAt(simTime()+150, spannerInterval);

        //The first time that the spanner is being build it is needed
to start sending data
        //after the first time it will be sent on itself.
        if(dataFirst)
        {
                sendData();
                dataFirst=false;
        }
        dispStr = getDisplayString();
        if(RSpanner>R)
        {
                dispStr.setTagArg("r",0,R);
        }
        else
        {
                dispStr.setTagArg("r",0,RSpanner+200);
        }
        if(ID==0 && !firstLead)
        {
            EV<<ID<<" is here"<<endl;
            firstLead = true;
            leaderInterval = new cMessage("leaderInterval");
            scheduleAt(simTime(), leaderInterval);

        }
        cVect_all_x.recordWithTimestamp(simTime(), position[0]);
        cVect_all_y.recordWithTimestamp(simTime(), position[1]);
        cVect_all_z.recordWithTimestamp(simTime(), position[2]);
        cHist_all_x.collect(position[0]);
        cHist_all_y.collect(position[1]);
        cHist_all_z.collect(position[2]);
        setDisplayString(dispStr);
}

/*********************************************************************
****************
 * Function name: sendData
 * Input: none
 * Output: void
 *
 * Description:This function send the heavy duty data messages that
are sent after
 *              all uavs are flying in group.
 *              This function distinguish between working in battery
saving mode
 *              using the spanner assigned transmission radius and the
regular mode
 *              that means to send in maximum range R.

*********************************************************************
*************/
void uav::sendData()
{
        //if there is no pre detected error that the data is sent.
        if(!err && !tooBig)
        {
            //Counting the number of times those kind off messages
where delivered.
            dataSent++;
            //Distinguishing between the two modes
            if(batterySaveMode)
            {
                //Using the battery saving mode will lead to using
RSpanner which is the
                //spanner defined transmission range that were
assigned to the uav.
                battery = battery - (RSpanner*RSpanner*P_D_ratio);
                cVect.recordWithTimestamp(simTime(),battery);
                cHist.collect(battery);
                //When the first uav's battery is extinguished then
the simulation is
                //terminated and the exact time of termination is
recorded.
                if(battery<=0)
                {
                    cVect2.record(simTime());
                    endSimulation();
                }
                //Scheduling the next data to be sent.
                data = new cMessage("data");
```

```
                        scheduleAt(simTime()+25, data);
                }
                else
                {
                        //Not using the battery saving mode will lead to 
using R which is the
                        //maximum transmission range that were defined.
                        battery = battery - (R*R*P_D_ratio);
                        cVect.recordWithTimestamp(simTime(),battery);
                        //When the first uav's battery is extinguished then 
the simulation is
                        //terminated and the exact time of termination is 
recorded.
                        cHist.collect(battery);
                        if(battery<=0)
                        {
                                cVect2.record(simTime());
                                endSimulation();
                        }
                        //Scheduling the next data to be sent.
                        data = new cMessage("data");
                        scheduleAt(simTime()+25, data);
                }
        }
        else
        {
                //if the uav has broke from the rest of the group (due to 
separation) than no
                //data is being sent.
                data = new cMessage("data");
                scheduleAt(simTime()+25, data);
        }
}

/*********************************************************************
****************
 * Function name: newPosition
 * Input: none
 * Output: void
 *
 * Description:This function will calculate and determine were 
exactly the uav's
 *              position and direction at the end of the current 
interval.
 *              The function uses the basic rules of "boids" such as 
alignment and
 *              cohision in order to determine those parameters.
 *              More detailed explanations of the way those rules are 
being executed
 *              can be found in the final report and also in the 
alignmentCalc and
 *              cohisionCalc functions descriptions.

**********************************************************************
**************/
void uav::newPosition()
{
        double *meanDv,*CohPos,*CohDv,CohDvSize,meanDvSize;
        double* Dv;
        double DvSize;
	//This part is not a part of the new position calculation. Here 
we check
	//if for the first time all the uavs are in a group and than we 
will start
	//to send data for the first time whether by a spanning tree or 
maximum
	//transmission range.
	if(neighborNum==maxUavs && firstTime)
	{
		firstTime=false;
		if(batterySaveMode)
		{
			EV<<"I'm here!!!!!!!!!"<<endl;
			bubble("Create Spanner");
			createSpanner();
		}
		else
		{
			sendData();
		}
	}


	// Alignment calculation
	//Receiving the alignment vector by calling the alignmentCalc 
function
	Dv = alignmentCalc();
	//Copying the vector into another array on which we will 
perform calculations
	meanDv = new double[3];
	meanDv[0] = Dv[0];
	meanDv[1] = Dv[1];
	meanDv[2] = Dv[2];
	//Calculating the vector size in order to normalize the values
	DvSize=sqrt(pow(meanDv[0],2)+pow(meanDv[1],2)+pow(meanDv[2],2))
;
	//Normalizing
	meanDv[0]=meanDv[0]/DvSize;
	meanDv[1]=meanDv[1]/DvSize;
	meanDv[2]=meanDv[2]/DvSize;
	//Calculating the normalized vector size
	meanDvSize=sqrt(pow(meanDv[0],2)+pow(meanDv[1],2)+pow(meanDv[2]
,2));
	//If we reached our goal than a new goul is being randomized in 
order to keep moving.
	if (DvSize <=  velocity)
	{
		diraction[0]= rand()%6000; // generates a new random 
destination
		diraction[1]= rand()%6000;
		diraction[2]= rand()%6000;
	}
	//If we haven't got to our goal than the new position is being 
calculated
	else
	{
		position[0]=position[0]+(meanDv[0]*velocity);
		position[1]=position[1]+(meanDv[1]*velocity);
		position[2]=position[2]+(meanDv[2]*velocity);
	}
	// Cohision calculation
	//Receiving the cohision vector by calling the cohisionCalc
function
	CohPos=cohisionCalc();
	if (CohPos[0] == -1)
	{
		//Copying the vector into another array on which we will
perform calculations
		CohDv = new double [3];
		CohDv[0]=CohPos[0];
		CohDv[1]=CohPos[1];
		CohDv[2]=CohPos[2];
		//Calculating the vector size in order to normalize the
values

	CohDvSize=sqrt(pow(CohDv[0],2)+pow(CohDv[1],2)+pow(CohDv[2],2))
;
		//Normalizing
		CohDv[0]=CohPos[0]/CohDvSize;
		CohDv[1]=CohPos[1]/CohDvSize;
		CohDv[2]=CohPos[2]/CohDvSize;
		//The new position is being calculated
		position[0]=position[0]+(CohDv[0]*velocity/cohVelFactor);
		position[1]=position[1]+(CohDv[1]*velocity/cohVelFactor);
		position[2]=position[2]+(CohDv[2]*velocity/cohVelFactor);
	}
	//After this interval is over a new interval is scheduled.
	interval = new cMessage("interval");
	scheduleAt(simTime(), interval);
}

/*********************************************************************
****************
 * Function name: newPosition
 * Input: none
 * Output: An array of three double values
 *
 * Description:This function calculate the cohision values of the
group.
 *                   UAVs within a flock are attracted to each other
as long as they are
 *             within the detection range, but outside the separation
range.
 *             This goal is to have the UAVs flock together, but not
to be so close
 *             that they are on top of each other.If there are too
many UAVs in the
 *             flock, the separation range will need to be increased.

*********************************************************************
**************/
double* uav::cohisionCalc()
{
	//Initializing the array in order to start summing up the
values
	double* PosSum =new double[3];
	PosSum[0]=0;
	PosSum[1]=0;
	PosSum[2]=0;
	//n is the number of current neighbors being counted and put 
into calculation.
	int n=0;
	//Summing up the positions of all neighbors
	for (int i=0; i<maxUavs; i++)
	{
		if(uavInfo[i].participate)
		{
			n=n+1;
			PosSum[0]= PosSum[0] + uavInfo[i].pos[0];
			PosSum[1]= PosSum[1] + uavInfo[i].pos[1];
			PosSum[2]= PosSum[2] + uavInfo[i].pos[2];
		}
	}
	if(n!=0)
	{
		//Dividing the sum in each dimension in the number of 
neighbors
		PosSum[0]= PosSum[0]/n;
		PosSum[1]= PosSum[1]/n;
		PosSum[2]= PosSum[2]/n;
	}
	else
	{
		PosSum[0]=-1;
	}
	//Returning the vector.
	return PosSum;
}

/*********************************************************************
****************
 * Function name: alignmentCalc
 * Input: none
 * Output: An array of three double values
 *
 * Description:This function calculate the alignment values of the 
group.
 *			In a flock of UAVs, each UAV will try to match 
the direction of the
 *			UAVs around it that it can detect. If some of 
the UAVs in a flock
 *		detect an obstacle (another UAV), they will turn.
 *		This causes the rest of the flock that can't even 
detect the
 *		obstacle to also turn away from it.

*********************************************************************
**************/
double* uav::alignmentCalc()
{
	//Initializing the array in order to start summing up the 
values
	double* DvSum =new double[3];
	DvSum[0]=0;
	DvSum[1]=0;
	DvSum[2]=0;
	//n is the number of current neighbors being counted and put 
into calculation.
	int n=0;
```

```
      int isParticipate[maxUavs];//array indicates the ignored
neighbors when computing alignment
    for(int i=0;i<maxUavs;i++)
      isParticipate[i]=1;
      //ignoringRandomNei(isParticipate);

      //Summing up the directions of all neighbors
            for (int i=0; i<maxUavs; i++)
            {
                  if(uavInfo[i].participate && isParticipate[i])
                  {
                        n=n+1;
                        DvSum[0]= DvSum[0] + uavInfo[i].dir[0];
                        //EV<<"Is participate: "<<i<<endl;
                        //EV<<"Before:"<<DvSum[0]<<endl;
                        DvSum[1]= DvSum[1] + uavInfo[i].dir[1];
                        DvSum[2]= DvSum[2] + uavInfo[i].dir[2];
                        addingDev(DvSum);
                        //EV<<"After:"<<DvSum[0]<<endl;
                  }
            }
      if(n!=0)
      {
            //Dividing the sum in each dimension in the number of
neighbors
            DvSum[0]= DvSum[0]/n;
            DvSum[1]= DvSum[1]/n;
            DvSum[2]= DvSum[2]/n;


            diraction[0]=DvSum[0];
            diraction[1]=DvSum[1];
            diraction[2]=DvSum[2];

            DvSum[0]=diraction[0]-position[0];
            DvSum[1]=diraction[1]-position[1];
            DvSum[2]=diraction[2]-position[2];
      }
      else
      {
            //In case no neighbors were found I shell countinue in my
current direction
            DvSum[0]=diraction[0]-position[0];
            DvSum[1]=diraction[1]-position[1];
            DvSum[2]=diraction[2]-position[2];
      }
      //Returning the vector.

      addingDevForMe(DvSum);

      return DvSum;
}

/*********************************************************************
****************
 * Function name: randomTime
 * Input: none
 * Output: void
 *
 * Description:This function randomize a small number which represent
the time
```

```
 *              that the UAV will wait in the begining of the current
interval
 *              in order to avoid some symmetry cases of which we
discussed in
 *              the final report.

*********************************************************************
**************/
void uav::randomTime()
{
      //Randomizing a small number
      int random_integer = rand()%10;
      //Creating rt message and scheduling it in the wanted time in
order to start
      //this interval calcultions.
      rt = new cMessage("randomTime");
      scheduleAt(simTime()+random_integer, rt);
}

/********************************************************************
****************
 * Function name: findNeigbors
 * Input: none
 * Output: void
 *
 * Description:This function is called right after the random time at
the beginning
 *              of the current interval and it creates neighborCheck
messages
 *              and broadcast them. At the end of this function we
schedule the
 *              message startCalc which start the calculations of
position in the current
 *              interval after waiting a specific amount of time to
receive replays
 *              from neighbors.

*********************************************************************
**************/
void uav:: findNeigbors()
{
      //Creating the message by calling the generateMessage function.
      neighborCheck = generateMessage((char*)("neighborCheck"));
      //Sending the message out of all outgoing gates
      for (int i=0; i<maxUavs; i++)
      {

            if (i != ID)
            {
                  cMessage *copy = neighborCheck->dup();
                  uavm *copy1 = check_and_cast<uavm *>(copy);
                  copy1->setDestination(i);
                  send(copy1, "g$o", i);
            }
      }
      //Deleting the message after sending
    delete neighborCheck;
    //we schedule the message startCalc which start the calculations
of
    //position in the current interval.
    // generates a message that will tell me when to start the calc
in order
    //to let the replay messages time to arrive
      startCalc = new cMessage("startCalc");
      scheduleAt(simTime()+20, startCalc);
}

/*********************************************************************
****************
 * Function name: initialPosition
 * Input: none
 * Output: void
 *
 * Description:This function sets the initial position for the UAV
and it is called
 *              from the initial message. Insert the initial position
from the
 *              display string

*********************************************************************
**************/
void uav::initialPosition()
{
      dispStr = getDisplayString();
      int random1 = rand()%6000;
      int random2 = rand()%6000;

      EV<<"MY ID is: "<<this->ID<<endl;
      position[0]=random1;
      position[1]=random2;
      EV<<"MY position: "<<this->position[0]<<endl;
      EV<<"MY position: "<<this->position[1]<<endl;
      dispStr.setTagArg("p",0,random1);
      dispStr.setTagArg("p",1,random2);
      dispStr.setTagArg("r",0,R);
      setDisplayString(dispStr);
      //After placing the UAV on the battlefield we call the first
interval by
      //producing an interval message and scheduling it to be
deployed next.
      interval = new cMessage("interval");
      scheduleAt(simTime(), interval);

}

/*********************************************************************
****************
 * Function name: initialPosition
 * Input: char *name
 * Output: uavm *uav
 *
 * Description:This function receives a char pointer which represent
the message name
 *              and create a message of type uavm message which
contain the UAV's
 *              id,position and direction and return this message.

*********************************************************************
**************/
uavm *uav::generateMessage(char *name)
{
    int src = getIndex();   // our module index

    uavm *msg = new uavm(name,0);
    msg->setSource(src);

    msg->setPosX(position[0]);
    msg->setPosY(position[1]);
    msg->setPosZ(position[2]);
    msg->setDirX(diraction[0]);
    msg->setDirY(diraction[1]);
    msg->setDirZ(diraction[2]);
    return msg;
}

/*********************************************************************
****************
 * Function name: initialPosition
 * Input: char *name
 *        int dest
 * Output: uavm *uav
 *
 * Description:This function receives a char pointer which represent
the message name
 *              and an int which represent the destination of the
message
 *              and create a message of type uavm message which
contain the UAV's
 *              id,position and direction and return this message.

*********************************************************************
**************/
uavm *uav::generateMessage(char *name,int dest)
{
      int src = getIndex();   // our module index
      int dst = dest;
      uavm *msg = new uavm(name,0);
      msg->setSource(src);
      msg->setDestination(dst);
      msg->setPosX(position[0]);
      msg->setPosY(position[1]);
      msg->setPosZ(position[2]);
      msg->setDirX(diraction[0]);
      msg->setDirY(diraction[1]);
      msg->setDirZ(diraction[2]);
      return msg;
}

/*********************************************************************
****************
 * Function name: separation
 * Input: int Id
 * Output: void
 *
 * Description:This function is called when the UAV get too close
(distance under
 *              the critical range C)to another UAV. The id of the
close UAV is
 *              given into the function (int Id).
 *              UAV's are repealed by other UAV's and by obstacles.
 *              This causes UAV to turn away from other UAVs.
```

```
***********************************************************************
**************/
void uav::separation(int Id)
{
	double Dv[3];//The current direction vector in the moment of
separation
	double aviodColVec[3];//The vector of avoidance - the direction
in which the
					//UAV would turn in order to avoid
	double colVec[3];//The vector of collide
	double aviodColVecSize;//The avoidance vector size
	double DvSize;//The size of the direction vector
	double colVecSize;//The collision vector size

	//Calculating the directional vector
	Dv[0] = diraction[0]-(double)position[0];
	Dv[1] = diraction[1]-(double)position[1];
	Dv[2] = diraction[2]-(double)position[2];

	//In case of same direction a default value is inserted
	if(Dv[0]==0)
	{
		Dv[0]=15;
	}
	if(Dv[1]==0)
	{
		Dv[1]=15;
	}
	if(Dv[2]==0)
	{
		Dv[2]=15;
	}

	//Calculating the collision vector
	colVec[0] = uavInfo[Id].fromHimtoMe[0];
	colVec[1] = uavInfo[Id].fromHimtoMe[1];
	colVec[2] = uavInfo[Id].fromHimtoMe[2];

	//In case of same direction a default value is inserted
	if(colVec[0]<0.1)
	{
		colVec[0]=0.1;
	}
	if(colVec[1]<0.1)
	{
		colVec[1]=0.1;
	}
	if(colVec[2]<0.1)
	{
		colVec[2]=0.1;
	}

	//Calculating the direction vector size
	DvSize = sqrt(pow(Dv[0],2)+pow(Dv[0],2)+pow(Dv[0],2));
	//Calculating the collision vector size
	colVecSize =
sqrt(pow(colVec[0],2)+pow(colVec[1],2)+pow(colVec[2],2));

	//Normalizing the directional vector
	Dv[0] = Dv[0] / DvSize;
	Dv[1] = Dv[1] / DvSize;
	Dv[2] = Dv[2] / DvSize;
	//Normalizing the collision vector
	colVec[0] = colVec[0] / colVecSize;
	colVec[1] = colVec[1] / colVecSize;
	colVec[2] = colVec[2] / colVecSize;

	//In case of same direction a default value is inserted
	if(Dv[0]<0.1)
	{
		Dv[0]=0.1;
	}
	if(Dv[1]<0.1)
	{
		Dv[1]=0.1;
	}
	if(Dv[2]<0.1)
	{
		Dv[2]=0.1;
	}

	if(colVec[0]<0.1)
	{
		colVec[0]=0.1;
	}
	if(colVec[1]<0.1)
	{
		colVec[1]=0.1;
	}
	if(colVec[2]<0.1)
	{
		colVec[2]=0.1;
	}

	//Calculating the matrix product vector
	aviodColVec[0]= Dv[1]*colVec[2]-Dv[2]*colVec[1]; // operator X
= vector multiplication
	aviodColVec[1]= Dv[2]*colVec[0]-Dv[0]*colVec[2];
	aviodColVec[2]= Dv[0]*colVec[1]-Dv[1]*colVec[0];

	//In case of same direction a default value is inserted
	if(aviodColVec[0]<0.1)
	{
		aviodColVec[0]=0.1;
	}
	if(aviodColVec[1]<0.1)
	{
		aviodColVec[1]=0.1;
	}
	if(aviodColVec[2]<0.1)
	{
		aviodColVec[2]=0.1;
	}

	//Calculating the avoidance vector size
	aviodColVecSize =
sqrt(pow(aviodColVec[0],2)+pow(aviodColVec[0],2)+pow(aviodColVec[0],2
));
	//Normalizing the avoidance vector
	aviodColVec[0]=  aviodColVec[0] / aviodColVecSize;
	aviodColVec[1]=  aviodColVec[1] / aviodColVecSize;
```

```
        aviodColVec[2]=  aviodColVec[2] / aviodColVecSize;

        //In case of same direction a default value is inserted
        if(aviodColVec[0]<0.1)
        {
              aviodColVec[0]=0.1;
        }
        if(aviodColVec[1]<0.1)
        {
              aviodColVec[1]=0.1;
        }
        if(aviodColVec[2]<0.1)
        {
              aviodColVec[2]=0.1;
        }

        //Calculating the new position after avoiding the UAV
        position[0]=position[0]+(aviodColVec[0]*velocity/cohVelFactor);
        position[1]=position[1]+(aviodColVec[1]*velocity/cohVelFactor);
        position[2]=position[2]+(aviodColVec[2]*velocity/cohVelFactor);

        if(position[0]>=6000)
        {
              position[0] = uavInfo[Id].pos[0]-2;
        }

        if(position[1]>=6000)
        {
              position[1] = uavInfo[Id].pos[1]-2;
        }

        if(position[2]>=6000)
        {
              position[2] = uavInfo[Id].pos[2]-2;
        }


        if(position[0]<=0)
        {
              position[0] = uavInfo[Id].pos[0]+2;
        }

        if(position[1]<=0)
        {
              position[1] = uavInfo[Id].pos[1]+2;
        }

        if(position[2]<=0)
        {
              position[2] = uavInfo[Id].pos[2]+2;
        }

 }
/**********************************************************************
****************
 * Function name: ignoringRandomNei
 *
 * Description: This function is called to randomly ignore
information from UAV's neighbors
 *
```

```
***********************************************************************
**************/
void uav::ignoringRandomNei(int isParticipate[])
{
	int numOfNei=0;
	int random=0;
	int smallRand=0;

		for (int i=0; i<maxUavs; i++)
		{
			if(uavInfo[i].participate)
			{
				numOfNei++;
				isParticipate[i]=1;//we insert 1 if is a 
neighbor
			}
			else
				isParticipate[i]=0;
		}
		random=(int)rand()%numOfNei;
		while(random!=0)//we ignore information from "random" 
random neighbors by putting 0 in the isParticipate array
		{
			smallRand=(int)rand()%maxUavs;
			if(isParticipate[smallRand])
			{
				isParticipate[smallRand]=0;
				bubble("Deleting someone...");
				EV<<"Deleting"<< 
smallRand<<"!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
<<endl;
				random--;
			}
		}

}
void uav::addingDev(double* DevSum)
{
	if((int)rand())
		DevSum[0]+=(int)rand()%100;
	else
		DevSum[0]-=(int)rand()%100;
	if((int)rand())
		DevSum[1]+=(int)rand()%100;
	else
		DevSum[1]-=(int)rand()%100;
	if((int)rand())
		DevSum[2]+=(int)rand()%100;
	else
		DevSum[2]-=(int)rand()%100;
}
void uav::addingDevForMe(double* DevSum)
{
	/*DevSum[0]+=(int)rand()%600;
	DevSum[1]+=(int)rand()%600;
	DevSum[2]+=(int)rand()%600;*/
	if((int)rand())
			DevSum[0]+=(int)rand()%600;
		else
			DevSum[0]-=(int)rand()%600;
```

```
            if((int)rand())
                  DevSum[1]+=(int)rand()%600;
            else
                  DevSum[1]-=(int)rand()%600;
            if((int)rand())
                  DevSum[2]+=(int)rand()%600;
            else
                  DevSum[2]-=(int)rand()%600;
}
void uav::startLead()
{
      //leaderInterval = new cMessage("leaderInterval");
      //scheduleAt(simTime()+1000, leaderInterval);

            imLead=true;
            EV<<"direction before: "<<diraction[0]<<endl;
            diraction[0]=rand()%6000;
            diraction[1]=rand()%6000;
            diraction[2]=rand()%6000;
            EV<<"direction after: "<<diraction[0]<<endl;
            saveConnectivity();

}
void uav::saveConnectivity()
{
      int sign_x;
      int sign_y;
      int sign_z;
      int dist;
      int move_x=0;
      int move_y=0;
      int move_z=0;
      int max_d=0;
      int d=0;
      bool fly=false;
      sign_x = diraction[0]-position[0];
      sign_y = diraction[1]-position[1];
      sign_z = diraction[2]-position[2];
      leaderMove = new cMessage("leaderMove");
      scheduleAt(simTime()+20, leaderMove);

      if(sign_x<0)
            move_x=-velocity/sqrt(3.0);
      else if(sign_x>0)
            move_x=velocity/sqrt(3.0);
      if(sign_y<0)
                  move_y=-velocity/sqrt(3.0);
            else if(sign_y>0)
                  move_y=velocity/sqrt(3.0);
      if(sign_z<0)
                  move_z=-velocity/sqrt(3.0);
            else if(sign_z>0)
                  move_z=velocity/sqrt(3.0);


      for(int j=0; j<maxUavs; j++)
      {
            EV<<"UAV "<<j<<"  direction: "<<uavInfo[j].dir[0]<<endl;
            EV<<"UAV "<<j<<"  position: "<<uavInfo[j].pos[0]<<endl;
      }
      for (int i=0; i<maxUavs; i++)
```

```
        {

                dist = sqrt(pow((position[0]+move_x) -
uavInfo[i].pos[0],2)+pow((position[1]+move_y)-
uavInfo[i].pos[1],2)+pow((position[2]+move_z)-uavInfo[i].pos[2],2));
                if(dist<=R)
                {
                        fly=true;
                        break;
                }

        }
        if(fly)
        {
                if(abs(diraction[0]-position[0])<=velocity)
                        position[0]+=diraction[0]-position[0];
                else
                        position[0]+=move_x;
                if(abs(diraction[1]-position[1])<=velocity)
                        position[1]+=diraction[1]-position[1];
                else
                        position[1]+=move_y;
                if(abs(diraction[2]-position[2])<=velocity)
                        position[2]+=diraction[2]-position[2];
                else
                        position[2]+=move_z;


        }
        else
        {
                /*if(abs(diraction[0]-position[0])<=velocity)
                        position[0]+=diraction[0]-position[0];
                else
                        position[0]+=move_x/4;
                if(abs(diraction[1]-position[1])<=velocity)
                        position[1]+=diraction[1]-position[1];
                else
                        position[1]+=move_y/4;
                if(abs(diraction[2]-position[2])<=velocity)
                        position[2]+=diraction[2]-position[2];
                else
                        position[2]+=move_z/4;*/
                for(int i=0; i<maxUavs; i++)
                {
                        d=sqrt(pow(R,2)-pow(uavInfo[i].pos[1]-
position[1],2)-pow(uavInfo[i].pos[2]-position[2],2))-
(abs(position[0]-uavInfo[i].pos[0]));
                        if(abs(d)>max_d)
                                max_d=abs(d);
                }
                if(sign_x<0)
                        if(max_d>velocity)
                                position[0]=position[0]-velocity;
                        else
                                position[0]=position[0]-max_d;
                else if(sign_x>0)
                                if(max_d>velocity)
                                        position[0]=position[0]+velocity;
                                else
                                        position[0]=position[0]+max_d;
        }
        EV<<"my direction: "<<diraction[0]<<endl;
        EV<<"my position is: "<<position[0]<<endl;

        dispStr = getDisplayString();
        dispStr.setTagArg("p",0,position[0]);
        dispStr.setTagArg("p",1,position[1]);
        dispStr.setTagArg("r",0,R);
        setDisplayString(dispStr);
        cVect_lead_x.recordWithTimestamp(simTime(),position[0]);
        cHist_lead_x.collect(position[0]);
        cVect_lead_y.recordWithTimestamp(simTime(),position[1]);
        cHist_lead_y.collect(position[1]);
        cVect_lead_z.recordWithTimestamp(simTime(),position[2]);
        cHist_lead_z.collect(position[2]);

        if(position[0]==diraction[0]&&position[1]==diraction[1]&&positi
on[2]==diraction[2])//||abs(diraction[1]-position[1])<=velocity)
        {
                diraction[0]=rand()%6000;
                diraction[1]=rand()%6000;
                diraction[2]=rand()%6000;
        }
        /*
        if(position[1]==diraction[1])//||abs(diraction[1]-
position[1])<=velocity)
                        diraction[1]=rand()%6000;
        if(position[2]==diraction[2])//||abs(diraction[1]-
position[1])<=velocity)
                        diraction[2]=rand()%6000;
        */

}

 void uav::finish()
 {
         cVect2.record(dataSent);
 }

}; //namespace
```

Bibliography

1	D. Gillen and D. Jaques, "Cooperative behavior schemes for improving the Effectiveness of Autonomous Wide Area Search Munitions" , Proceedings of the Cooperative Control Workshop, 2000.
2	P. Chandler and M.pachter, "Heirarchical Control for Autonomous Teams", AIAA Guidance, Navigation, and Control Conference and Exhibit, 2001.
3	J. Hebert, "Cooperative Control of UAV's'', AIAA Guidance, Navigation, and Control Conference and Exhibit, 2001.
4	M. Polycarpou, "A Cooperative Search Framework for Distributed Agents'', Proceedings of the 2001 IEEE International Symposium on Intelligent Control , 2001.
5	L. Parker, "Current State of the Art in Distributed Autonomous Mobile Robotics", Autonomous Robotic Systems4, edited by L. E. Parker, pp. 3-12. Springer-Verlag, 2000.
6	S. Dolev, S. Gilbert, N. A. Lynch, E. Schiller, A. Shvartsman, J. Welch, ``Virtual Mobile Nodes for Mobile Ad Hoc Networks'' International Conference on Principles of DIStributed Computing (DISC 2004), 2004. Also Brief announcement in Proc. of the 23th Annual ACM Symp. on Principles of Distributed Computing, (PODC 2004), 2004
7	S. Dolev, S. Gilbert, N. A. Lynch, A. Shvartsman, J. Welch, ``GeoQuorum: Implementing Atomic Memory in Ad Hoc Networks'' 17th International Conference on Principles of DIStributed Computing, Springer-Verlag LNCS:2848, (DISC 2003), pp. 306-320, 2003.
8	O. Ben-Shahar and S.W. Zucker , The Perceptual Organization of Texture Flows: A Contextual Inference Approach, In the IEEE Transaction on Pattern Analysis and Machine Intelligence 25(4), 401-417, 2003. http://www.cs.bgu.ac.il/~ben-shahar/Publications/PAMI2003-POCV.pdf

9	E.W.Dijekstra, "Self stabilizing systems in spite of distributed control", Communication of the ACM, vol.17, 1974, pp. 643-644.
10	S.Dolev, Self-Stabilization, MIT Press, 2000.
11	S.Dolev, J.L.Welch, "Random Walk for Self-Stabilizing Clock Synchronization in the presence of Byzantine Faults," journal of the ACM,Vol.51, No.5, pp. 780-799, September 2004.
12	A.Daliot, D.Dolev, and H.Parnas, "Linear Time Byzantine Self-Stabilizing Clock Synchronization", Proc. Of the 7th international conference on principles of Distributed Systems(OPODIS 2003),2003.
13	"NOVA Online". NOVA. 1999. http://www.pbs.org/wgbh/nova/bees/dancesroun.html. Retrieved 2008-12-21.
14	G.Beni, J.Wang, "Swarm Intelligence in Cellular Robotic Systems", Proc. NATO Advanced Workshop on Robots and Biological Systems, Tuscany, Italy, June 26–30 (1989).
15	X. Y.Li, G.Calinescu, P.J. Wan, "Distrebuted Construction of Planner and Routing for Ad Hoc Networks", IEEE INFOCOM, 2002. http://www.cs.bgu.ac.il/~segal/148.pdf
16	Riley, J. R. et al. (12 May 2005) The flight paths of honeybees recruited by the waggle dance. Nature 435, pp. 205-207. doi:10.1038/nature03526
17	Seeley, T.D., P.K. Visscher, and K.M. Passino. (2006) Group decision making in honey bee swarms. American Scientist. 94:220-229.
18	Frisch, Karl von. (1967) The Dance Language and Orientation of Bees. Cambridge, Mass.: The Belknap Press of Harvard University Press.
19	Thom et al. (21 August 2007) The Scent of the Waggle Dance. PLoS Biology. Vol. 5, No. 9, e228 doi:10.1371/journal.pbio.0050228[1]
20	Bozic J., C. Abramson, M. Bedencic. (April 2006) Reduced ability of ethanol drinkers for social communication in honeybees (Apis mellifera carnica Poll.). Alcohol. Volume 38 , Issue 3. pp. 179-183.

21	http://news.bbc.co.uk/earth/hi/earth_news/newsid_8176000/8176878.stm BBC News 31 July 2009 Honeybees warn of risky flowers Matt Walker
22	Abbott, Kevin; Reuven Dukasa (23 July 2009). "Honeybees consider flower danger in their waggle dance". Animal Behaviour. http://dx.doi.org/10.1016/j.anbehav.2009.05.029. Retrieved 01-08-2009.
23	Frisch, Karl von. (1967) The Dance Language and Orientation of Bees. Cambridge, Mass.: The Belknap Press of Harvard University Press.
24	Frisch, Karl von. (1967) The Dance Language and Orientation of Bees. Cambridge, Mass.: The Belknap Press of Harvard University Press.
25	Aristotle, Historia animalium, IX, 40, Becker 624b; modified from the translation by D.W. Thompson in The Works of Aristotle, Clarendon, Oxford, 1910.