Robotic Swarm Networks: A Low-Cost Alternative Robotic Solution for Surveillance, Documentation, and Resource Gathering

Ryan Gadsby, May 2010

Boston College Computer Science Departmen

ABSTRACT

From the surface of Mars, to terrestrial car factories, robots play an important role in many aspects of human life. However, the limitations of robotic hardware – power consumption, imprecise movement, and monetary cost chief among them - are often a massive obstacle to the implementation of many possible applications of this technology. Swarm robotics provides several methods of circumventing these roadblocks. Smaller, more primitive robots typically consume less power and cost less money to develop than larger, more intricate machines. However, by distributing tasks among multiple smaller robots, the same amount of work can be accomplished as if we designated the same task to a single rover. The problem of imprecise movement can be alleviated by using multiple robots to interpolate their estimated surroundings with existing data instead of relying on a single machine's interpretation of its environments. By using a large network consisting of low-cost robots, we can take a more cavalier approach to the use of robots of exploration and reconnaissance: since even an inexpensive sensor on an expensive machine might be irreplaceable and therefore limit the possible applications of a given robot, a member of a swarm can disappear without inconveniencing its brothers unduly.

This thesis examines several different swarm network configurations and their performance at tasks analogous to real-world applications of swarm technology. This is achieved using a low-cost modular robotic controller developed by LEGO Mindstorms known as the NXT, and installing a Java Virtual Machine on it to provide a hassle-free development platform. The benefits of swarm technology are fully explored in the swarm's implementation as their collective sensory input is used to form a “hive mind” accessible by any given member of the swarm.

I focused on the task of searching for an object in the physical environment, and compared the time spent and general effectiveness of the swarm to a single bot equipped with more sensors doing the same task.

INTRODUCTION

I. The growing role of robots in human society

Humanity has come to rely on robots in many aspects of society. In the industrial sector, we have intricate, precise, robotic installations designed to either manufacture or inspect various goods, such as automobiles. In the commercial sector, floor-crawling rovers vacuum floors, freely available to anyone with the inclination and disposable income to do so. On the surface of Mars, the twin observational rovers Spirit and Opportunity scour the dusty red landscape, transferring the results of their examinations to NASA geologists. The U.S. Military deploys unmanned aircraft for the purposes of reconnaissance without risking human life in the process. Surgical robots are employed to aid doctors with delicate operations. Even everyday consumers can purchase a Roomba robot to methodically vacuum their houses with no additional input on their part besides turning it on. Robots are capable of performing tasks unsuitable or too dangerous for humans to directly get involved with, and can even be called upon to undertake tasks that humans deem too tedious or menial to waste human effort on. However, there are prominent difficulties that arise where robotic technology is concerned that must be addressed in any attempt to harness its benefits. This is especially true when considering mobile robots like rovers and walkers, the primary types of automata used in swarm applications of robotic technology.

In January of 2004, NASA sent two robotic rovers to the surface of Mars – Spirit and Opportunity. For the past six years they have been gathering vital geological data that will be invaluable for any attempts to terraform the desert-like surface of the red planet. They have even found potential evidence of former water bodies or even microbial life. Relying on robots to do a

job too costly, dangerous, and time-consuming for humans to perform is one of the foremost applications of robotic technology. (Mars Exploration Rover project)

Robot-assisted or completely unmanned surgery has become a way for doctors to reliably perform different operations with a degree of precision impossible to duplicate by human hands alone. Robots can delicately perform minimally invasive tasks that human hands cannot, simply because of the smaller size and locations of interaction that can be handled by machine. Robotic surgical assistants are now frequently used in neurological, cardiological, and orthopedic surgeries where even small amounts of error are intolerable. (Ahmed K; Khan MS; Vats A; Nagpal K; Priest O; Patel V; Vecht JA; Ashrafian H, 2009)

The Roomba, an autonomous vacuuming robot designed for ordinary consumers has become incredibly popular. As of September 2009, manufacturer iRobot has sold over 3 million robots. Perhaps more importantly, thanks to this considerable presence in the consumer sector, many Roomba users have taken an interest in robotics, going to far as to "hack" them with the aid of introductory programming books available at any bookstore. This growing trend of humans life becoming intertwined with robots even in everyday life is indicative of the advances made in robotic technology over the past few decades. (http://www.irobot.com/sp.cfm?pageid=203)

II. Swarm robotics as opposed to conventional robotics

Swarm robotics is the application of many small, primitive robots to a problem as opposed to using a single robot. Its study is derived from the inner workings and characteristics of social insect swarms such as ants or bees. One of its goals is to replicate the swarm intelligence that is naturally found in these organisms, and using this aspect to produce various useful behaviors. For example, a "hive mind" can be formed by members of a swarm sharing

their sensory data with each other. (K.A.Hawick, H.A.James, J.E.Story and R.G.Shepherd, 2007)

Applications for swarm robotics present themselves at many levels of technology. Where primitive swarm members are appropriate for doing a basic physical task - such as mining, surveying, or foraging – swarms can be a preferable alternative to using human labor due to health risks or general tedium. Where miniaturization is a factor, swarm robotics can be applied to nanotechnology or micromachinery to handle distributed sensory tasks in the human body. By decentralizing intelligence we allow for more primitive swarm members, instead relying on parallel computation to perform tasks using macroscopic control over the entire swarm. As a whole, however, swarm robotics has yet to emerge outside of the research sector, and there are no current commercial, military, or recreational implementations of it. (Waldner, Jean-Baptiste , 2007)

Current developments in swarm robotics include JASMINE, an open-source effort in coordinating emulating biological swarm behavior using robots smaller than 3cm by 3cm. The JASMINE robots themselves are incredibly primitive, sending wireless messages to each other on rudimentary Bluetooth chips. For sensors, each is equipped with an array of IR devices that serve as eyes. For locomotion the JASMINE robot only has two tiny electric servos, each one moving a single wheel. These robots aren't truly capable of meaningful interaction with the physical world, and are instead used to simulate and examine swarm behavior from a biological standpoint. They can be compared in terms of complexity to either single cells such as members of a mold fungus. (Schmickl and Crailsheim, 2006)

SYMBRION is another swarm configuration, based on swarm members sharing physical resources by docking together to form structures and share energy. It explores the notion of artificial evolution to see what behaviors the swarm will develop to accomplish a given task. SYMBRION's entire platform is built on the concept of pervasive adaptiveness. When one

swarm member (or a subset of the swarm) 'figures out' how to accomplish something, the entire swarm is able to capitalize upon this knowledge. This combination of simulating social insect behavior with artificial evolution allows SYMBION swarms to perform tasks such as linking up to form a bridge to transport virtual 'resources' or even members of the swarm itself. (Ficici, Watson, Pollack, 1999)

REPLICATOR, a swarm designed to be fully self-sufficient by exhibiting self-assembling and self-programming behavior. The individual robotic modules are larger and heavier than in JASMINE or SYMBRION, but the same principles of interconnectedness and adaptation are applied. REPLICATOR is more focused on creating sensor- and communication-rich modules capable of assembling and programming new swarm members instead of relying on a fixed number of swarm members or outside assistance for a given task. It is similar in configuration to SYMBRION insofar that it relies on adaptive behavior to direct the overall intelligence of the swarm, but the approach is different as a result of the greater complexity of individual swarm members. (Jason Teo, 2004)

.

III. Problems with conventional robotic technology

The use of mobile robots – robots not permanently installed like factory assemblers – carry in tow a significant number of problems that must be addressed to have their application to any given problem worth the trouble. Mobile robots are not typically attached to any constant power source, and so they are subject to the limitations of the whatever battery they are equipped with. The movements of rovers and walkers tend to be rather imprecise and unreliable, making it rather difficult to maneuver a robot to a specific, absolute location. Due to the time required to design, calibrate, and then program and test these robots for whatever tasks and environments

they are required to encounter, robots tend to be rather costly to engineer in terms of time and, subsequently, currency. As a result of these costs, it often becomes difficult to justify subjecting these machines to harsh environments that may damage them. While addressable, these concerns must weigh heavily in the mind of any engineer before making any attempt to approach a problem from a robotic point of view.

Power consumption is one of the largest barriers to effectively applying robotic technology to humanity's problems. Batteries simply do not last long enough to power a robot's motors and sensors for many tasks, especially those which last a long time such as exploration. A common solution is to use solar panels, such as on the Mars rovers Spirit and Opportunity, but solar panels don't generate enough energy to keep robots functioning constantly, and certain environmental factors can render them useless altogether. Robots need access either to a constant energy source or intermittent contact with an energy source in order to maintain function as much as possible. Robotic swarm networks have methods of circumventing this restriction. (Mei, Lu, Hu, Lee 2005)

The electric motors utilized on rovers and walkers lead to problems with imprecise movement that stunt robots' usefulness when it comes to reconnaissance and charting an area. Even motors with tachometers that can specify precisely what angle to move the motor to can be foiled by a change of topography, or uneven terrain. Conventional methods to circumvent this problem involve using some sort of absolute locating method such as GPS, but this only helps a robot know where it is, not where a given set of motor movements will take it. Some sort of reckoning needs to be enforced, as the collective error accumulated by imprecise motor movements will cause a robot to get “lost” very quickly. (Chung, Ojeda, Borenstein, 2001)

Another problem that must be overcome in order to find a reasonable robotic solution to a given task is the cost to design, produce, and program a given robot. Once the actual hardware is

produced (after great care is taken to select the right array of motors and sensors to accomplish the task at hand), a great deal of time is required to calibrate and program the robot to perform a task as aptly as a human would. And even then, the final product of all this time and money is a robot that handles a very specific task. Often we must make a decision of making a robot that performs a single or limited subset of tasks as well as or better than a human, or making a robot that can be applied less adeptly at a greater range of endeavors.

As a result of these imposing costs, we further limit the applications of robots in “unsafe” environment. A team of engineers who spent fifteen million dollars on a rover would certainly be reluctant to send it to observe a lava flow, especially when humans are quick-footed enough to do it themselves without undue risk. Why program an automated unmanned airborne vehicle when we can simply remote-control it and eliminate losing large financial investments due to an uncaught buffer overflow? For the most part, building large intricate machines leaves us completing tasks with robots that humans could accomplish as well or better, but are simply too menial or tedious.

IV. The benefits of a robotic swarm as compared to a conventional robot

Luckily, swarm robotics provides many methods to circumvent these restrictions. By spreading work out over many robots instead of using just one robot, we can distribute power consumption in a manner that doesn't require the swarm to stop doing work. By using multiple robots, we can use multiple perspectives as an additional form of reckoning to alleviate the problem of imprecise mobility. Smaller, more primitive robots are less expensive to manufacture; as a result individual swarm members are expendable and can therefore be sent to perform tasks that could result in their incapacitation or destruction more readily.

In terms of power consumption, robotic swarms tend to have many advantages when compared to larger, more complicated robots. Their smaller, more primitive motors, sensors, and CPUs simply use less energy, although they are also typically equipped with smaller batteries. However, since the swarm is composed of many smaller units, allowing one to rest and recharge via solar panels or some sort of energy source allows the swarm to continue its work relatively unimpeded. The collective capacity to do work before recharging in a swarm will be greater than that done by a single robot, and work will always still be done even as individual members have entered an inactive mode in order to recharge their batteries. (Schmickl and Crailsheim, 2006)

Imprecise movement is still a problem for robotic swarms, but swarms are better equipped to meliorate the impact of the issue than singular robots. By being able to pinpoint or even estimate each robot's position relative to its brothers, we can construct a much more reliable model of a given robotic operation than relying on one robot's impressions of its surroundings. Multiple cameras can be used to create a stereoptic analysis of a system, and the collective estimations therein can be averaged to provide a realistic if imperfect snapshot of the environment the swarm is deployed to. This way, we can have a proficient system of reckoning without having to rely on a costly absolute system such as GPS or radar. (Moeslinger, Schmickl and Crailsheim, 2009)

The emphasis on using many smaller, less sophisticated robots to perform a task in swarm robotics is often a more cost-effective solution than having one or two intricate machines. Swarm members are typically not designed with durability or multi-tasking in mind, and can therefore be equipped with inexpensive sensors, motors, and CPUs. Some sensors can become very expensive as higher qualities are necessary, such as cameras and infrared detectors, but we need not equip every member of the swarm similarly. Indeed, using the sensor network of the entire swarm, it is possible to have robots not equipped with sensors at all – individually blind, but directed by the

"hive-mind" of the swarm. (Schmickl and Crailsheim, 2006)

Because of the low cost of individual machines, members of a swarm can be viewed as expendable, allowing them to perform tasks that one might think twice about using a single expensive robot for. While we might be hesitant to send a multi-million dollar automated submersible to examine undersea volcanic vents, a member of a swarm can perform this task ably with a cheap camera and whatever relevant sensors while transmitting any data back to his brothers before getting destroyed. This allows us to completely automate certain tasks that are normally left to humans themselves – or at least humans remotely controlling robots – without worry.

Aside from providing methods to overcome the main issues with mobile robotic technology, swarm robotics gives us ways to perform tasks that would simply be infeasible for normal robots. Robots tend to move sluggishly, and where time is an issue their employment is often unattractive. One or two robots, no matter how sophisticated, are not an optimal way to search for survivors in collapsed mine, to sniff out explosive devices in a sprawling urban landscape, or to alert soldiers of intruders along a wide perimeter. A swarm, made of many subunits, can cover ground much more quickly and survey a greater area, casting a wide sensor net to find its objectives. The area covered by having multiple nodes contribute to the "hive mind" of the entire swarm is a vital resource when time is a factor.

V. The drawbacks of using swarm technology

Swarm technology, however, is not without its drawbacks. Having to account for multiple interconnected robots introduces many implementation difficulties, such as accounting for missing nodes and needing to calibrate the sensors of many robots individually. As a result, the

number of variables that could cause something to go horribly wrong increase greatly, and must be accounted for to have any reasonable application of swarm technology. Using more primitive sensors and motors can also degrade the quality of the work done by the swarm. They are not suitable for undertaking tasks that require very specific observations or sensory input, like exhaustively exploring an entire area and surveying it to produce an accurate map.

With a single robot, one must only account for one set of sensors, one CPU, and one body of code. This is sadly not so in a swarm. By having multiple robots equipped with a variety of sensors on a robot-by-robot basis, there are several different protocols that must be accounted for as the individual machines interact. In a completely networked approach this raises a great number of possible interactions that must be taken into account, and even in a structured, hierarchical swarm a different set of code is necessary for each level of command. Moreover, simply having robots rely on each other for awareness of their surrounding raises several networking issues that must be addressed. If we are assuming a node can disappear at any time – expendability being one of the main reasons to employ a swarm – we must also provide a swarm with the means to restructure itself when one of its members becomes absent.

Because there are so many robots in a swarm, there are a great many more variables that must be accounted for when implementing any sort of swarm application. Unless all of the robots in the swarm have the exact same hardware configuration, there will need to be multiple bodies of code that must be designed, implemented, and debugged separately. Even if we encapsulate all of the common elements of the swarm into one code base, the problems that arrive from the different sets of sensors or motor configurations can lead to many problems. Time spent debugging is increased dramatically as each build has to be uploaded to every bot in the swarm. For example, it's easy to misdiagnose the odd behavior of one robot as a hardware fault when it was simply overlooked while uploading the latest build.

Because the physical hardware swarm robots are equipped with tend not to be terribly sophisticated, more problems arise. The lack of high-quality sensors and motors can make swarms unsuitable for tasks that require detailed sensory feedback or delicate movement. For example, equipping every robot in the swarm with a high resolution, telescopic lens camera defeats the purpose of using a swarm as a cost-effective alternative to a single sophisticated robot. They are better suited to tasks that merely require imprecise calculation: searching for the general location of a resource, or estimating the depth of a cave, for example.

METHODOLOGY

1. The NXT Brick from LEGO Mindstorms

All swarm implementation was done in Java using ten Lego NXT "bricks" equipped with leJOS firmware. These relatively inexpensive computers come in packages with modular, interchangeable sensors and motors that made it easy to test different configurations. Moreover, a third-party camera designed for the NXT was also used. Ten skeletal robots were assembled, and several different network models were tested.

The NXT Brick itself possesses an ARM7 microprocessor clocked at the relatively unimpressive speed of 46 megahertz and a meager cache of 32 kilobytes. While this is more than enough computing power to handle basic motor, sensor, and communication routines, it severely limits any sort of complicated data processing that can be done by the swarm without relying on an external source. Luckily, the distributed nature of the swarm helps alleviate this drawback, as computational tasks can be relegated to inactive members.

NXT Bricks can currently only interface with one type of motor, which is included in the Mindstorms kit. It's an electric servo motor with a high degree of motion accuracy thanks to its on-board tachometers: accurate to within one degree of specificity. Any sort of locomotive task – either moving the robot itself or manipulating an object – was undertaken using one or more of these motors.

The Mindstorms kit comes with a variety of sensors, and many third-party sensors have been developed for the NXT brick. Included are:

- A touch sensor that simply gives binary feedback for touch or release.
- A light sensor that gives quantitative feedback of light intensity overall, or the

light intensity of certain colors.

- A sound sensor that gives quantitative levels of sound in either dB or dBA.
- An ultrasonic sensor that approximates the distance of an external object by emitting an infrared “ping”.
- An accelerometer that measures the rotational position of the robot.
- A compass sensor to tell the heading of the robot with regards to magnetic north.
- An RFID sensor that gives feedback to match a given RFID frequency.

Moreover, cameras have been developed via third parities to give the robot a way of visually interfacing with its environment. For this thesis, the NXTCam developed by <> was used for this purpose. The proprietary sensors used were limited to the ones included in the Mindstorms kit: the touch, light, sound, and ultrasonic sensors. This is because of the cost associated with extra sensors whose function was either limited for the scope of this thesis or could readily be duplicated with the other sensors.

The NXT brick has seven ports designed for 4-conductor cables to connect to: three for motors labeled A,B, and C, and four for sensors with numeric labels. To keep the solution of cost-effectiveness in mind, most of the bricks were equipped with only 2 motors and 2 or less sensors. Many members of the swarm were not equipped with sensors at all. This kept each member's functionality at bare minimal levels in order to demonstrate the power of the swarm as a whole.

The NXT bricks have important limitations that accentuate the problem of using more primitive hardware for a swarm as compared to a single robot. The most drastic limitation by far is the connection limit of the NXT's on-board Bluecore chip that handles Bluetooth communication. A given NXT brick may only maintain three connections at once, whether they are to other NXT bricks or other Bluetooth devices such as a GPS receiver or a laptop. As such,

swarm structures were inherently limited to hierarchical configurations rather than fully-connected graphs that a swarm would ideally possess. This could be worked around by deleting and reinstating connections at runtime, but the significant increase in running time wasn't justified when any data could be relayed through the hierarchy more quickly with the proper network implementation.

The other limitation was the memory capacity of the NXT brick. Each brick only has 256K of flash memory, more than half of which was taken up by the leJOS firmware. This limited not only the amount of Java classes that could be used in implementing the swarm's behavior, but also the amount of persistent image data that could be relayed back to an external source.

II. The leJOS custom firmware

The leJOS firmware is an open source project that enables the ability to put a Java Virtual Machine on each NXT brick. This portable version of the JVM greatly eases the networking issues of robotic implementation by allowing for the opportunity to keep code as high-level as possible. Although other custom NXT firmware platforms were available, leJOS is currently the only one that allows direct access to the NXT's Bluecore Bluetooth communications chip, which was irreplaceable in allowing the individual members of the swarm to communicate with each other.

leJOS can handle code for any class in the Java API as well as its own API that has classes implemented for different kinds of motors and sensors. The code for a given class is included at compile-time, so it's not hard-coded into the meager amount of memory used by the firmware. Since we only have 256K worth of flash memory on the NXT – nearly half of it being

used up by the firmware itself – steps must be taken to conserve the remaining amount of memory carefully. As such, it often is a smarter solution to use predefined arrays instead of relying on generic Java List classes.

III. The CommandCenter Abstract Class

The abstract class that I designed to encapsulate swarm behavior is called CommandCenter. It handles all of the communication between members of the swarm and is responsible for making sure all messages and commands go to their intended recipients. It is attached in its entirety in the appendix. Its design is that of a union of state machines that take input from each other, so that once initiated the entire swarm is fully automated until its task is complete or – in the case of a task that doesn't necessarily have a completion condition – shut down externally.

CommandCenter – an abstract class – has any communication-related methods implemented directly in the class. Subclasses, however, must implement swarm behavior specifically by overriding the abstract method executeCommand(). In a loop, this method essentially acts as state transition by using input from another robot to govern its own behavior to put it in a different state, such as DOCUMENT_LOCATION, INVESTIGATE_POSITION, or PERFORM_PHYSICAL_TASK. Three separate swarm configurations were implemented across three separate levels of command, and so nine separate subclasses of CommandCenter were implemented.

Each robot in the swarm is initialized with a x/y location and a facing vector. These values are maintained by a MovementCenter object that each CommandCenter subclass object has its own instance of. These positions are predetermined as there's no simple way to change

these at runtime thanks to the leJOS's lack of command line interface. The top member of the swarm hierarchy maintains a list of all positions of members of the swarm, and the swarm is initialized via the masters' requests for every member's position. After this, the swarm begins to act autonomously, although behavior varies by swarm configuration.

The master maintains three separate connections to the mid-level members of the swarm, and by alternating these it receives input for his CommandCenter state machine. The general format for any commands or date sent are an operation code followed by a series of parameters, usually a source/target robot, a location to move to, or even image data. The overall design of swarm behavior is to make sure the master can force as much work as possible onto his subordinates, who in turn find subordinates who are currently not doing anything. So, instead of having a robot with a camera spend time processing image data, it instead distributes each color channel to a separate subordinate so that it may continue on to the next objective point and take more pictures. In a hierarchical swarm this process is absolutely necessary to keep the top-level robot by being completely inundated with work.

For example, if a subcommander or leaf robot receives a DOCUMENT_LOCATION signal from its superior, it then reads two more integer values – the target X and Y positions for the target location. After individually deciding how to approach that position and taking a picture, it reports back to its commander before re-entering the READY state, waiting for input from its superior once more.

Each level of command had a different subclass of CommandCenter, although each command level uses the same subclass in a given swarm configuration. The uniformity therein makes each level much easier to manage. There is a different type of CommandCenter structure for each swarm configuration as well. As stated, this resulted in nine total subclasses of CommandCenter.

Each CommandCenter has an object called MovementCenter that track's the robots facing and position and handles each robot's movement subroutines. It can be calibrated for any given surface, and uses a Cartesian grid system relative to the starting position of the master robot in the swarm. It can be told to go to a point, or to simply face a point. The amount of rotation and movement are handled by using simple Cartesian distance formulas to calculate the distance, and uses vector products to calculate the difference in degrees between the target facing and the current facing. There are also several extra methods for moving away from a given point, or to simply approach a point (which is useful for swarms equipped with cameras, since taking a picture when the robot is at the point where the objective is supposed to be is often somewhat less than useful.

IV. Swarm configuration Triforce

The first sensor configuration for the ten-strong swarm I used for this thesis is called “Triforce”. Triforce features three camera-equipped robots that are directly subservient to the root commander of the swarm. At the leaves of the swarm are 6 sensorless bricks that are used primarily to store and calculate image data from the three camera robots. Relevant conclusions (image blob location, etc) are sent back upwards through the swarm network and stored on the leaves for later access by an external source. While equipped with motors, they are primarily immobile, but can be used to interact with the environment on a basic level (such as holding a position or pushing an object.) This leaves the root node to act as the sole investigator, looking for appropriate areas or objects to send the cameras to using the ultrasonic sensor and the light sensor. This multiple-camera setup can be used to simulate binocular (or trinocular) vision to approximate real-world distance to an object, or to give a fuller perspective on a given object. It

could also be used to simply document an object or an environment from multiple perspectives for the sake of completeness or redundancy of information.

V. Swarm configuration Parker

The second sensor configuration for the swarm is called “Parker”. It works essentially as an inversion of the Triforce configuration. The root node is equipped with camera instead of the second command level, and the second command level function as investigators using the light and ultrasonic sensors. As in Triforce, the leaf robots are used to process data and perform simple mundane tasks, as they are sensorless. Most importantly, however, they are used to relay objective data to outside sources, alerting them to the position and condition of a given objective. Compared to Triforce, Parker's strength lies in the ability to cover ground to find potential objectives to document as quickly as possible. It is akin to a search-and-rescue or resource-locating swarm as opposed to the observational nature of the Triforce configuration.

VI. Swarm configuration Gaga

Gaga, the last swarm configuration, differs from Triforce and Parker in that it is designed to accomplish work by physically interacting with its environment. Here, the leaf robots are used as worker drones to accomplish work. Its overall sensor configuration is similar to Parker: the root node has a camera, and each of the subcommanders has a ultrasonic sensor and a light sensor used to find objectives. The leaf node behavior is entirely different, however. Rather than be used simply for calculating data from their respective parents, Gaga's leaf robots travel to locations of interest designated by the swarm and simulate performing a physical task. This

swarm configuration is best-suited to a wide-scale environmental task such as taking soil samples or agricultural foraging.

VII. The tasks

One task was chosen for each swarm configuration. The time taken to complete each task was recorded for both the corresponding swarm and a single bot equipped to do each task on its own. All of the "objectives" used in these tasks are simply red plastic cups placed at random locations in the environment that the swarm (or single robot) explores. Three timed trials for each swarm were taken.

The task for Triforce involves documenting an objective from several perspectives. For the Triforce swarm, the root node is used to locate these objectives while the subcommanders simultaneously document it using cameras. For the analogous single robot, the robot must do all of the documentation from different perspectives himself. Leaf robots were used for the purpose of simulating physical work.

The task for Parker involves thoroughly documenting and processing information about objectives. The Parker root node handles documentation while the subcommanders handle detection of objectives. Computation and processing this information is done by the leaf nodes. In the corresponding single bot, all of this work – detection, documentation, and processing is done by a single machine. Because image data can't be transferred using the NXTCam at runtime due to the IC2 sensor port restrictions, dummy images of the NXTCam resolution was processed to account for run-time. Capture time and transfer time are still taken into account for the swarm. The data is then relayed to an external source, in keeping with Parker's theme of quickly reporting objective locations for search-and-rescue style tasks.

The task for Gaga is similar to Parker's, but instead physical work must be performed on the objectives. To simulate these actions, any robots defined as workers spin in place for 30 seconds at the objective to demonstrate a physical task, such as drilling or gathering a sample. The singe robot must wait until this work is completed before it can continue searching for another objective. In all other aspects, at the root commander and subcommander level, the task is the same as that undertaken by the Parker swarm.

RESULTS

I. Multi-perspective documentation with a single investigator

Trial Number	Swarm Time	Single Robot Time
1	1:46	3:32
2	1:53	3:56
3	2:38	3:28
4	---	3:36
5	2:07	3:45

II. Single-perspective documentation with multiple investigators

Trial Number	Swarm Time	Single Robot Time
1	1:23	1:53
2	1:34	2:12
3	2:01	2:34
4	1:46	2:59
5	1:16	1:57

III. Environment interaction with multiple investigators

Trial Number	Swarm Time	Single Robot Time
1	2:38	4:57
2	---	4:01
3	---	5:12
4	2:45	4:53
5	2:26	4:33

IV. Conclusions

The general trend with the NXT swarms was that they were consistently able to perform their given tasks significantly faster than single robots. However, this only applies when the tasks themselves were completed. Due to problems with pathing, reckoning, synchronization and communication, colliding swarm members inhibited the swarm's ability to actually finish the task at hand.

Parker, however, was unaffected by this as its immobile leaf robots and inherently divergent subcommander movement patterns prevented any collisions. Triforce and Gaga, due to their more active leaf robots, were more likely to experience a fatal crash that rendered their task not completable.

This problem could have been alleviated by more efficient pathing algorithms, or allowing the swarm to give a more thorough estimation of its current density at given locations as environmental features to avoid.

REFERENCES

- Mars Exploration Rover project, NASA/JPL document NSS ISDC 2001 27/05/2001

- Ahmed K; Khan MS; Vats A; Nagpal K; Priest O; Patel V; Vecht JA; Ashrafian H; et al. (Oct 2009). Int J Surg. 7:431-440

- K.A.Hawick, H.A.James, J.E.Story and R.G.Shepherd, *An Architecture for Swarm Robots* p2-3

- Waldner, Jean-Baptiste (2007). *Nanocomputers and Swarm Intelligence*. London: ISTE. pp. 242-p248

- Schmickl and Crailsheim, *A navigational algorithm for swarm robotics inspired by slime mold aggregation,* Second SAB 2006

- Sevan G. Ficici, Richard A. Watson, Jordan B. Pollack, *Embodied Evolution: A Response to Challenges in Evolutionary Robotics,* Eighth European Workshop on Learning Robots, 1999

- Jason Teo, *Darwin + Robots = Evolutionary Robotics: Challenges in Automatic Robot Synthesis ,* 2nd International Conference on Artificial Intelligence in Engineering and Technology [ICAIET 2004], volume 1, pages 7-13, Kota Kinabalu, Sabah, Malaysia, August 2004

- Yongguo Mei, Yung-Hsiang Lu, Y. Charlie Hu, and C.S. George Lee: *A Case Study of*

Mobile Robot's Energy Consumption and Conservation Techniques , pp4-5, 2005

- Hakyoung Chung , Lauro Ojeda, and Johann Borenstein, *Sensor fusion for Mobile Robot Dead-reckoning With a Precision-calibrated Fiber Optic Gyroscope* , pp2-3, 2001 IEEE International Conference on Robotics and Automation, Seoul, Korea, May 21-26, pp. 3588-3593

- Christoph Moeslinger1 , Thomas Schmickl1 , and Karl Crailsheim1: *A Minimalist Flocking Algorithm for Swarm Robots, 2009*

APPENDIX

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

abstract class CommandCenter{
	protected List<BTConnection> con;
	protected List<DataInputStream> in;
	protected List<DataOutputStream> out;

	public int botID;

	protected MovementCenter thisBot;

	public static final int BUSY=240;
	public static final int NO_CHANGE=241;
	public static final int CONFIRMED = 242;

	public static final int INITIALIZE_LOCATIONS=250;
	public static final int INVESTIGATE = 251;
	public static final int DOCUMENTED = 252;
	public static final int REPORT = 253;
	public static final int WORK = 254;


	public CommandCenter(int id, MovementCenter bot){
		con = new ArrayList<BTConnection>();
		in = new ArrayList<DataInputStream>();
		out = new ArrayList<DataOutputStream>();
		botID = id;
		thisBot = bot;
	}

	public void addConnection(BTConnection btc){
		if (con.size()<3){
			con.add(btc);
			in.add(btc.openDataInputStream());
			out.add(btc.openDataOutputStream());
		}
	}
	abstract void executeCommand() throws IOException,InterruptedException;
```

```
	public void initialize() throws IOException{
		//botLocation.add(botID, new Point(thisBot.getX(),thisBot.getY()));

		//connect to parent
		addConnection(Bluetooth.waitForConnection());
		//connect to children
		for (int i = 1; i<3;i++){
			RemoteDevice rd = Bluetooth.getKnownDevice("Gadsby" +(i+botID));
			addConnection(Bluetooth.connect(rd));
		}
		//good to go
		ready = true;
	}
	public boolean isReady(){
		return ready;
	}

}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class TriforceCommand extends CommandCenter {
	private UltrasonicSensor ultra;
	private LightSensor light;
	private Random gen;

	private List<Point> botLocation;

	private boolean autonomous;

	public TriforceCommand(int id, MovementCenter bot){
		super(id,bot);
		gen = new Random();
		ultra = new UltrasonicSensor(SensorPort.S1);
		light = new LightSensor(SensorPort.S3);
		botLocation = new ArrayList<Point>();
		autonomous=true;
		ready =true;
	}


	public void executeCommand() throws IOException, InterruptedException{
		if(autonomous){//explore, then send
			Thread.sleep(200); //let sensors warm up
			if (ultra.getDistance() < 160 || light.readValue()>40){
				Sound.playTone(1760,1000);
				//found something, send notify children of coordinates!
				this.requestInvestigation();
			}
			int dX = gen.nextInt(3);
			int dY = gen.nextInt(3);
			if (gen.nextInt()%2==1)
				dX = dX*-1;
			if (gen.nextInt()%2==1)
				dY = dY*-1;
			if (!locationOccupied(thisBot.getX()+dX,thisBot.getY()+dY))
				thisBot.goToPoint(thisBot.getX()+dX,thisBot.getY()+dY);

		}
		else{ //get data from children, interpret and act accordingly.
```

```
                this.waitForDocumentation();
            }
        }

        private void requestPositionUpdates() throws IOException{
            for (int i = 0;i<3;i++){
                try{
                    DataInputStream dis = in.get(i);
                    DataOutputStream dos = out.get(i);

                    dos.writeInt(CommandCenter.INITIALIZE_LOCATIONS);
                    dos.flush();
                    LCD.drawInt(i,i,i);
                    int response = dis.readInt();
                    LCD.drawInt(response,4,4);
                    if (response != CommandCenter.BUSY && response !=
CommandCenter.NO_CHANGE){
                        //get botID and point, add it to locations.
                        botLocation.add(dis.readInt(),new
Point(dis.readInt(),dis.readInt()));
                        dos.writeInt(CommandCenter.CONFIRMED);
                    }
                }catch(IOException e){
                        LCD.drawInt(34,0,0);
                }
            }
        }

        private void requestInvestigation()throws IOException, InterruptedException{
            for (int i = 0; i<3;i++){
                out.get(i).writeInt(CommandCenter.INVESTIGATE);
                out.get(i).writeInt(thisBot.getX());
                out.get(i).writeInt(thisBot.getY());
                out.get(i).flush();
            }
            //move away from average swarm location
            Point p = averageLocation();
            thisBot.moveAwayFromPoint(p.x,p.y,4);
            autonomous = false;
        }

        private Point averageLocation(){
            int x=0;
            int y = 0;
            for (Point p : botLocation){
                x += p.x;
                y += p.y;
```

```
            }
            return new Point(x/botLocation.size(),y/botLocation.size());
        }

        private boolean locationOccupied(int x, int y){
            for (Point p : botLocation){
                if (x == p.x || y == p.y)
                    return true;
            }
            return false;
        }

        private void waitForDocumentation()throws IOException{
            while (true){
                for (int i = 0; i<3;i++){
                    int c = in.get(i).readInt();
                    if (c == CommandCenter.DOCUMENTED){
                        LCD.drawString("Bot " + i + " finished",i,i);
                    }
                }
                Button.waitForPress();
            }
        }
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class TriforceSubcommand extends CommandCenter{
	private NXTCam cam;

	private boolean selfreported;
	private boolean child1reported;
	private boolean child2reported;

	public TriforceSubcommand(int id, MovementCenter bot){
		super(id,bot);
		cam = new NXTCam(SensorPort.S1);
		ready =false;
		selfreported = false;
		child1reported = false;
		child2reported = false;
	}

	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				int command = in.get(0).readInt();

				if (command == CommandCenter.INITIALIZE_LOCATIONS)
					initializeLocations();
				if (command == CommandCenter.INVESTIGATE)
					investigate();
			}catch (IOException e){
			}
		}
	}

	private void initializeLocations() throws IOException{
		LCD.drawInt(0,0,0);
		if (!selfreported){
			out.get(0).writeInt(0);
			out.get(0).writeInt(botID);
			out.get(0).writeInt(thisBot.getX());
			out.get(0).writeInt(thisBot.getY());
			out.get(0).flush();
			selfreported=true;
```

```
                }
                else if (!child1reported){
                        out.get(0).writeInt(0);
                        out.get(0).writeInt(botID);
                        out.get(0).writeInt(thisBot.getX());
                        out.get(0).writeInt(thisBot.getY());
                        out.get(0).flush();
                        child2reported=true;
                }
                else if (!child2reported){
                        out.get(0).writeInt(0);
                        out.get(0).writeInt(botID);
                        out.get(0).writeInt(thisBot.getX());
                        out.get(0).writeInt(thisBot.getY());
                        out.get(0).flush();
                        child2reported=true;
                }
                else{
                        out.get(0).writeInt(CommandCenter.NO_CHANGE);
                        out.get(0).flush();
                }
                if (in.get(0).readInt() != CommandCenter.CONFIRMED) //something went wrong
                        Sound.playTone(1760,1000);
        }

        private void investigate() throws IOException,InterruptedException{
                while (true){
                        try{
                                int x = in.get(0).readInt();
                                int y = in.get(0).readInt();

                                //go to location
                                thisBot.goToPoint(x,y);
                                //add camera stuff here
                                out.get(0).writeInt(CommandCenter.DOCUMENTED);
                                out.get(0).flush();
                                break;
                        }catch(IOException e){
                                                //Thread.sleep(500);
                        }
                }
        }
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class TriforceLeaf extends CommandCenter{
	public TriforceLeaf(int id, MovementCenter bot){
		super(id,bot);
	}
	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				int command = in.get(0).readInt();
				if (command == CommandCenter.INITIALIZE_LOCATIONS)
					initializeLocations();
				if (command == CommandCenter.INVESTIGATE)
					investigate();
			}catch (IOException e){
			}
		}
	}

	public void initialize() throws IOException{
		addConnection(Bluetooth.waitForConnection());
		ready = true;
	}

	private void initializeLocations() throws IOException{
			out.get(0).writeInt(0);
			out.get(0).writeInt(botID);
			out.get(0).writeInt(thisBot.getX());
			out.get(0).writeInt(thisBot.getY());
			out.get(0).flush();
	}









	private void investigate() throws IOException,InterruptedException{
```

```
                while (true){
                        try{
                                int x = in.get(0).readInt();
                                int y = in.get(0).readInt();

                                //go to location
                                thisBot.goToPoint(x,y);
                                out.get(0).writeInt(CommandCenter.DOCUMENTED);
                                out.get(0).flush();
                                break;
                        }catch(IOException e){
                                                        //Thread.sleep(500);
                        }
                }
        }
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import java.util.*;

public class MovementCenter{
	public static final int LATERAL_CONSTANT = 1500;
	public static final double RADIAL_CONSTANT = .13;

	private int posX;
	private int posY;
	private float dirX;
	private float dirY;

	public static void main(String[] args) throws InterruptedException{
		MovementCenter thisBot = new MovementCenter(0,0,0f,1.0f);
		Thread.sleep(3000); //3 second delay
		thisBot.moveToFace(1,1);
		thisBot.moveToFace(-1,-1);
		thisBot.approachPoint(4,3);
		thisBot.approachPoint(0,0);
	}

	public MovementCenter(int px, int py, float dx, float dy){
		Motor.A.setPower(50);
		Motor.C.setPower(50);
		this.posX = px;
		this.posY = py;
		this.dirX = dx;
		this.dirY = dy;
	}


	public void move(float munits) throws InterruptedException{
		//positive is forward. Motors A and C are wheels.
		if (munits>0){
			Motor.A.forward();
			Motor.C.forward();
		}
		if (munits<0){
			Motor.A.backward();
			Motor.C.backward();
			munits = munits * -1;
		}

		Thread.sleep(Math.round(munits*LATERAL_CONSTANT));
		Motor.A.stop();
```

```
		Motor.C.stop();
		//forced wait to keep motors from bucking with constant movement
		Thread.sleep(200);
	}

	public void rotate(int degrees) throws InterruptedException{
		//positive is counterclockwise
		if (degrees>0){
			Motor.A.forward();
			Motor.C.backward();
		}
		if (degrees<0){
			Motor.A.backward();
			Motor.C.forward();
			degrees = degrees * -1;
		}
		Thread.sleep((long)Math.round(50*degrees*RADIAL_CONSTANT));
		Motor.A.stop();
		Motor.C.stop();
		Thread.sleep(200);
	}

	public void goToPoint(int x, int y) throws InterruptedException{

		float v2x = x - posX;
		float v2y = y - posY;
		int angle = (int)Math.round(Math.toDegrees(Math.atan2(v2y,v2x) -
Math.atan2(dirY,dirX)));
		this.rotate(angle);
		float distance = (float)Math.sqrt((x-posX)*(x-posX)+(y-posY)*(y-posY));
		this.move(distance);
		posX = x;
		posY= y;
		//make new direction vector via projection of onto destination
		float length =(float)Math.sqrt(v2x*v2x+v2y*v2y);
		dirX= v2x/length;
		dirY= v2y/length;
	}

	public void moveToFace(int x, int y) throws InterruptedException{
		//create vector, calculate angle
		float v2x = x - posX;
		float v2y = y - posY;
		int angle = (int)Math.round(Math.toDegrees(Math.atan2(v2y,v2x) -
Math.atan2(dirY,dirX)));

		this.rotate(angle);
```

```
		//make new direction vector via projection of onto destination
		float length =(float)Math.sqrt(v2x*v2x+v2y*v2y);
		dirX= v2x/length;
		dirY= v2y/length;
	}

	public int getX(){
		return posX;
	}
	public int getY(){
		return posY;
	}

	public void approachPoint(int x, int y)throws InterruptedException{//get within 3 units of
target
			int dx = this.posX - x;
			int dy = this.posY - y;
			int targetX=x, targetY=y;
			if (dx > 0)
				targetX = x+1;
			if (dx < 0)
				targetX = x-1;
			if (dy > 0)
				targetY = y+1;
			if (dy < 0)
				targetY = y-1;

			this.goToPoint(targetX,targetY);
			this.moveToFace(x,y);
		}

		public void moveAwayFromPoint(int x, int y, float distance) throws
InterruptedException{
			//create vector, calculate angle and distance.
			float v2x = x - posX;
			float v2y = y - posY;

			int angle = 180 - (int)Math.round(Math.toDegrees(Math.atan2(v2y,v2x) -
Math.atan2(dirY,dirX)));

			this.rotate(angle);
			this.move(distance);

			posX = x;
			posY= y;
			//make new direction vector via projection of onto destination
			float length =(float)Math.sqrt(v2x*v2x+v2y*v2y);
```

```
                    dirX= v2x/length;
                    dirY= v2y/length;
            }
    }
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class ParkerRootCommand extends CommandCenter{

	private List<Point> botLocation;
	private List<Rectangle> rects;
	private NXTCam camera;


	public ParkerRootCommand(int id, MovementCenter bot){
		camera = new NXTCam(SensorPort.S1);
		botLocation = new ArrayList<Point>();
		rects = new ArrayList<Rectangle>();
		super(id,bot);
	}

	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				for (int i = 0; i<3;i++){ //alternate between channels
					int command = in.get(i).readInt();
					if (command == CommandCenter.INVESTIGATE)
						investigate();
					else
						reportResults();
				}
			}catch (IOException e){
			}
		}
	}
	private void reportResults() throws IOException{
		//write results back to computer
		con.get(0).close(); //because of three-connection limit
		BTConnection c = Bluetooth.waitForConnection();
		DataOutputStream dos = c.openDataOutputStream();
		for (Rectangle rect:rects){
			dos.writeInt(rect.x);
			dos.writeInt(rect.y);
			dos.writeInt(rect.width);
			dos.writeInt(rect.height);
			dos.flush();
```

```
		}
	}

	private void requestPositionUpdates() throws IOException{
			for (int i = 0;i<3;i++){
				try{
					DataInputStream dis = in.get(i);
					DataOutputStream dos = out.get(i);

					dos.writeInt(CommandCenter.INITIALIZE_LOCATIONS);
					dos.flush();
					LCD.drawInt(i,i,i);
					int response = dis.readInt();
					LCD.drawInt(response,4,4);
					if (response != CommandCenter.BUSY && response !=
CommandCenter.NO_CHANGE){
						//get botID and point, add it to locations.
						botLocation.add(dis.readInt(),new
Point(dis.readInt(),dis.readInt()));
						dos.writeInt(CommandCenter.CONFIRMED);
					}
				}catch(IOException e){
						LCD.drawInt(34,0,0);
				}
			}
	}

	private void investigate(int i) throws IOException,InterruptedException{
		while (true){
			try{
				int x = in.get(i).readInt();
				int y = in.get(i).readInt();

				//go to location
				thisBot.goToPoint(x,y);
				//camera captures rect information and saves for later
				Rectangle rect = cam.getRectangle(0);
				rects.add(new Rectangle(thisBot.getX(),thisBot.getY(),
rect.height,rect.weight); //records size and location of each tracked object
				break;
			}catch(IOException e){}
		}
	}
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class ParkerSubcommand extends CommandCenter{
	private UltrasonicSensor ultra;
	private LightSensor light;
	private Random gen;

	private boolean autonomous;
	private boolean ready;
	private int botID;

	public ParkerSubcommand(int id, MovementCenter bot){
		gen = new Random();
		ultra = new UltrasonicSensor(SensorPort.S1);
		light = new LightSensor(SensorPort.S3);
		autonomous=true;
		super(id,bot);
	}



	public void executeCommand() throws IOException, InterruptedException{
		if(autonomous){//explore, then send
			Thread.sleep(200); //let sensors warm up
			if (ultra.getDistance() < 160 || light.readValue()>40){
				Sound.playTone(1760,1000);
				//found something, send notify parent of coordinates!
				this.requestInvestigation();
			}
			int dX = gen.nextInt(3);
			int dY = gen.nextInt(3);
			if (gen.nextInt()%2==1)
				dX = dX*-1;
			if (gen.nextInt()%2==1)
				dY = dY*-1;
			thisBot.goToPoint(thisBot.getX()+dX,thisBot.getY()+dY);

		}
	}
```

```
	private void requestInvestigation()throws IOException, InterruptedException{
		out.get(0).writeInt(CommandCenter.INVESTIGATE);
		out.get(0).writeInt(thisBot.getX());
		out.get(0).writeInt(thisBot.getY());
		out.get(0).flush();

		out.get(1).writeInt(CommandCenter.REPORT);
		out.get(1).writeInt(thisBot.getX());
		out.get(1).writeInt(thisBot.getY());
		out.get(1).flush();
		out.get(2).writeInt(CommandCenter.REPORT);
		out.get(2).writeInt(thisBot.getX());
		out.get(2).writeInt(thisBot.getY());
		out.get(2).flush();
	}
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class ParkerLeaf extends CommandCenter{

	public ParkerLeaf(int id, MovementCenter bot){
		selfreported = false;
		super(id,bot);
	}

	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				int command = in.get(0).readInt();

				if (command == CommandCenter.INITIALIZE_LOCATIONS)
					initializeLocations();
				if (command == CommandCenter.REPORT)
					report();
			}catch (IOException e){
			}
		}
	}

	public void initialize() throws IOException{
		//connect to parent
		addConnection(Bluetooth.waitForConnection());
		ready = true;
	}

	private void initializeLocations() throws IOException{
			out.get(0).writeInt(0);
			out.get(0).writeInt(botID);
			out.get(0).writeInt(thisBot.getX());
			out.get(0).writeInt(thisBot.getY());
			out.get(0).flush();
	}

	private void report() throws IOException,InterruptedException{
		while (true){
			try{
				int x = in.get(0).readInt();
```

```
                    int y = in.get(0).readInt();

                    addConnection(Bluetooth.waitForConnection());
                    in.get(1).writeInt(x);
                    in.get(1).writeInt(y);
                }catch(IOException e){
                                        //Thread.sleep(500);
                }
            }
        }
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class GagaRootCommand extends CommandCenter{

	private List<Point> botLocation;
	private List<Rectangle> rects;
	private NXTCam camera;

	public GagaRootCommand(int id, MovementCenter bot){
		camera = new NXTCam(SensorPort.S1);
		botLocation = new ArrayList<Point>();
		rects = new ArrayList<Rectangle>();
		super(id,bot);
	}

	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				for (int i = 0; i<3;i++){ //alternate between channels
					int command = in.get(i).readInt();
					if (command == CommandCenter.INVESTIGATE)
						investigate();
					else
						reportResults();
				}
			}catch (IOException e){
			}
		}
	}
	private void reportResults() throws IOException{
		//write results back to computer
		con.get(0).close(); //because of three-connection limit
		BTConnection c = Bluetooth.waitForConnection();
		DataOutputStream dos = c.openDataOutputStream();
		for (Rectangle rect:rects){
			dos.writeInt(rect.x);
			dos.writeInt(rect.y);
			dos.writeInt(rect.width);
			dos.writeInt(rect.height);
			dos.flush();
		}
```

```
	}

	private void requestPositionUpdates() throws IOException{
			for (int i = 0;i<3;i++){
				try{
					DataInputStream dis = in.get(i);
					DataOutputStream dos = out.get(i);

					dos.writeInt(CommandCenter.INITIALIZE_LOCATIONS);
					dos.flush();
					LCD.drawInt(i,i,i);
					int response = dis.readInt();
					LCD.drawInt(response,4,4);
					if (response != CommandCenter.BUSY && response !=
CommandCenter.NO_CHANGE){
						//get botID and point, add it to locations.
						botLocation.add(dis.readInt(),new
Point(dis.readInt(),dis.readInt()));
						dos.writeInt(CommandCenter.CONFIRMED);
					}
				}catch(IOException e){
						LCD.drawInt(34,0,0);
				}
			}
	}

	private void investigate(int i) throws IOException,InterruptedException{
		while (true){
			try{
				int x = in.get(i).readInt();
				int y = in.get(i).readInt();

				//go to location
				thisBot.goToPoint(x,y);
				//camera captures rect information and saves for later
				Rectangle rect = cam.getRectangle(0);
				rects.add(new Rectangle(thisBot.getX(),thisBot.getY(),
rect.height,rect.weight); //records size and location of each tracked object
				break;
			}catch(IOException e){}
		}
	}
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class  GagaSubcommand extends CommandCenter{
	private UltrasonicSensor ultra;
	private LightSensor light;
	private Random gen;

	private boolean autonomous;
	private boolean ready;
	private int botID;

	public GagaSubcommand(int id, MovementCenter bot){
		gen = new Random();
		ultra = new UltrasonicSensor(SensorPort.S1);
		light = new LightSensor(SensorPort.S3);
		autonomous=true;
		super(id,bot);
	}



	public void executeCommand() throws IOException, InterruptedException{
		if(autonomous){//explore, then send
			Thread.sleep(200); //let sensors warm up
			if (ultra.getDistance() < 160 || light.readValue()>40){
				Sound.playTone(1760,1000);
				//found something, send notify parent of coordinates!
				this.requestInvestigation();
			}
			int dX = gen.nextInt(3);
			int dY = gen.nextInt(3);
			if (gen.nextInt()%2==1)
				dX = dX*-1;
			if (gen.nextInt()%2==1)
				dY = dY*-1;
			thisBot.goToPoint(thisBot.getX()+dX,thisBot.getY()+dY);

		}
	}
```

```
	private void requestInvestigation()throws IOException, InterruptedException{
		out.get(0).writeInt(CommandCenter.INVESTIGATE);
		out.get(0).writeInt(thisBot.getX());
		out.get(0).writeInt(thisBot.getY());
		out.get(0).flush();
		//send workers
		out.get(1).writeInt(CommandCenter.WORK);
		out.get(1).writeInt(thisBot.getX());
		out.get(1).writeInt(thisBot.getY());
		out.get(1).flush();
		out.get(2).writeInt(CommandCenter.WORK);
		out.get(2).writeInt(thisBot.getX());
		out.get(2).writeInt(thisBot.getY());
		out.get(2).flush();
	}
}
```

```
import lejos.nxt.*;
import lejos.nxt.comm.*;
import lejos.nxt.addon.*;
import javax.bluetooth.*;
import java.io.*;
import java.util.*;
import java.awt.Point;

public class GagaLeaf extends CommandCenter{

	public GagaLeaf(int id, MovementCenter bot){
		super(id,bot);
	}
	public void executeCommand() throws IOException,InterruptedException{
		while (true){
			try{
				int command = in.get(0).readInt();

				if (command == CommandCenter.INITIALIZE_LOCATIONS)
					initializeLocations();
				if (command == CommandCenter.WORK)
					work();
			}catch (IOException e){
			}
		}
	}

	public void initialize() throws IOException{
		//connect to parent
		addConnection(Bluetooth.waitForConnection());
		ready = true;
	}

	private void initializeLocations() throws IOException{
			out.get(0).writeInt(0);
			out.get(0).writeInt(botID);
			out.get(0).writeInt(thisBot.getX());
			out.get(0).writeInt(thisBot.getY());
			out.get(0).flush();
	}

	private void work() throws IOException,InterruptedException{
		while (true){
			try{
				int x = in.get(0).readInt();
				int y = in.get(0).readInt();
```

```
                    //go to location
                    thisBot.goToPoint(x,y);
                    out.get(0).writeInt(CONFIRMED);
                    out.get(0).flush();
                    thisBot.rotate(3600);
                    break;
                }catch(IOException e){
                                        //Thread.sleep(500);
                }
            }
        }
}
```