The Swarm Orchestra: Temporal Synchronization and Spatial Division of Labor for Large Swarms of Autonomous Robots

James McLurkin

MIT Artificial Intelligence Laboratory
200 Technology Square, Cambridge, MA 02139
jamesm@ai.mit.edu

iRobot
22 McGrath Highway Suite 6, Somerville, MA 02143, U.S.A.
jamesm@irobot.com

Abstract

There are many important and useful tasks you can do with a swarm of 100 autonomous robots working together. Playing music in an orchestra is not one of these applications, but its two main building blocks, temporal synchronization and spatial division of labor, can be used to construct more serious applications. The robotic orchestra described in this paper is comprised of 35 robots. Our distributed algorithms allow them to synchronize their internal timers, and "clump" into groups based in the instrument they are currently playing. The net result is a swarm of robots playing music together, spatially organized such that robots playing the same instrument are near each other.

Table of Contents

1. Introduction

The goal of the iRobot Swarm project is to develop distributed algorithms for robotic swarms composed of hundreds of individual robots. Ultimately, we want to be able to write software for large number of robots at the group level. The software development system would then compile these group programs into software for the individual robots to run. This top-down problem is very difficult to solve, so we are approaching it from a bottom-up perspective. By making group behavior building blocks that can be recombined and reused for many different global applications, we will develop a swarm programming toolkit that can be used to construct complex global behaviors.

The SwarmBot

The iRobot SwarmBot shown in Figure 2 has been designed from the ground up for developing distributed algorithms on real hardware in large swarms. It has a high-performance 32-bit microprocessor, good mobility, and inter-robot communication and localization with the ISIS™ infrared communication system. The ISIS system allows each robot to determine the bearing, orientation, and range of its neighbors. Each robot is 5" on a side, and the total swarm has 100 members.

SwarmOS

The Swarm Operating System (SwarmOS) provides the foundation for the distributed algorithms. It provides low-level robot control, ISIS drivers, a virtual pheromone messaging system, a set-based neighbor query system, and remote programming and debugging facilities.

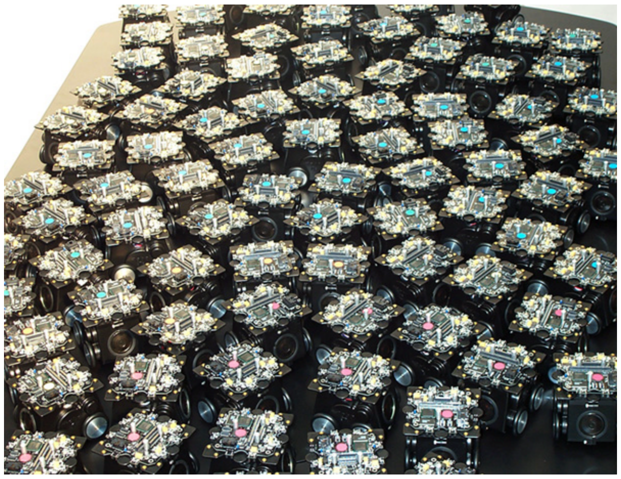


Figure 1: The iRobot Swarm is composed of 100 individual robots that work together to accomplish group goals. The swarm was developed with grants from DARPA under the Software for Distributed robots program.

Distributed Algorithm Behavior Library

Our distributed algorithms are comprised of many behaviors working together. Group actions result from the interactions of the behaviors of each robot, and the interactions of the robots with each other. We have a large library of these behaviors, and have used them to perform group tasks such as: clustering to a location of interest, dispersing to search a room, surrounding an object to measure its perimeter, and navigating long distances using neighboring robots as landmarks.

Hands-Off Operation: HIVE™ and the Robot Ecology™

To work with a large swarm of robots effectively, the user cannot manually program, charge, or even turn on the robots. Software development, debugging, and analysis must also be performed in a centralized fashion, without having to interact with each robot. The Robot Ecology™ provides resources the robots need to keep themselves running, and the HIVE user interface provides centralized command and control of the swarm. For large swarms, these are requirements, not luxuries.

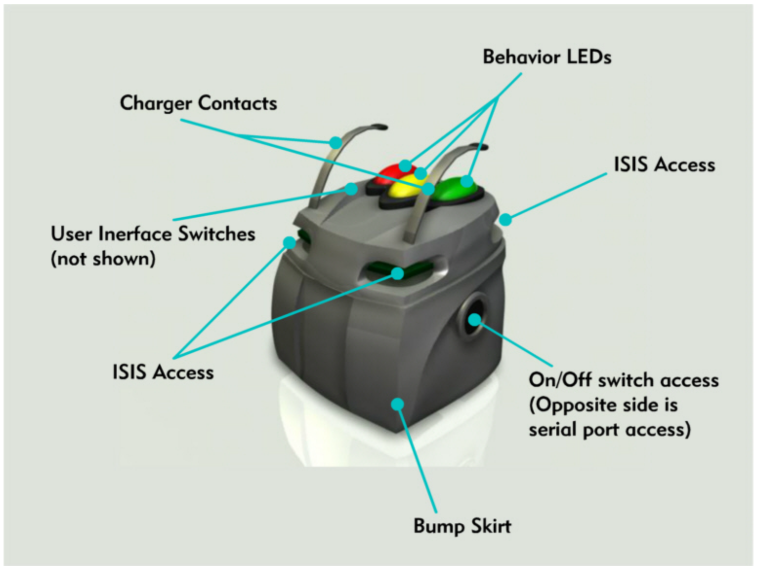


Figure 2:The iRobot SwarmBot™ has been designed from the ground up for embodied distributed algorithm development. It packs a comprehensive sensor suite, inter-robot communication and localization with the ISIS™ system, a 32-bit microprocessor, and a rugged low-maintenance design into a 5" cube that can rest in the palm of your hand.

2. The Swarm Orchestra

Each SwarmBot has a 1.1 watt audio system based on the Yamaha YM2413 FM-synthesis sound chip. This devise has nine audio channels, 16 different instrument sounds, and five percussion sounds. The SwarmOS has a general MIDI interpreter and patch table to play standard MIDI files. This capability is usually used for debugging software. By using the sound carefully, the software engineer can hear what the entire swarm is doing all at once, and then focus her attention on robots of interest, or robots that are misbehaving. But all work and no play make the SwarmBot a dull robot, so this debugging tool has been hijacked for our amusement…

Assumptions

- The music will be in a MIDI file, which the robots can play using their standard sound output drivers. All the robots have the same MIDI file. If there are multiple songs in the repertoire, the robots will be explicitly told which one to play before hand. (no improvisation!)
- The algorithms will only use local ($r \approx 3$ feet) communications from robot to robot. However, the user will be allowed to interact with the swarm as a whole to download music and software.
- The user will be allowed to interact with one robot, the "mediumBot", to initiate behaviors (clumping, temporal synchronization, and music playback). However, this robot will only be allowed to communicate with its immediate neighbors.

- The processor boards have reasonable crystal oscillators, which will limit drift in between temporal synchronizations.
- Each robot has a unique identification number. This is used extensively to disambiguate communications signals from neighboring robots.

3. MIDI File Playback

The SwarmOS has drivers for playing general MIDI files. The entire general MIDI spec has not been implemented (Pitch bends, etc…) but the basic note messages are sufficient to play most music. This playback mechanism was modified to play individual tracks masked by a 32-bit bit vector to allow playback of specific tracks from the song. The LED outputs were tied to the sound drivers, giving a display of the current audio output level.[1] This is useful for debugging, and for dramatic[2] visualizations of the performance.

4. Temporal Synchronization

In order to play music together (that is pleasing to humans (in particular, <u>this</u> human)), the robots will require some form of temporal synchronization. The direct approach, to transmit a global sync signal to all the robots simultaneously, is disallowed by our assumptions, so robots will have to propagate local sync signals from one to another. Distributed synchronization has been studied in natural systems [Mirollo 90]. These systems share information with sync pulses, and use the onset and frequency of these pulses to synchronize. The SwarmBots transmit 64 bits of information in each communication packet, which is enough space to encode the actual time from one source robot to share with the others.

Our algorithm relies on a temporal leader, whom all the other robots in the swarm synchronize to. The leader becomes a source for the temporal sync pheromone. This leader can be selected at random, but in our implementation, it is convenient to use the mediumBot, because it is the only robot we are able to interact with. This lets us stop and start synchronization for debugging, and read status updates from the swarm.

Because the propagation of the pheromone signal is a breadth-first search of the communications graph, all other robots will have at least one neighbor that is one hop closer to the source than they are. Each robot will sync to one of these neighbors. This ensures that all the robots sync towards the source, and eliminates cycles that could lead to unbounded accumulation of error.

Temporal Synchronization Algorithm

The robots all periodically broadcast neighbor information to each other. This is the ISIS neighbor update cycle, and it occurs every 250 ms. The information is used to determine the

[1] Special thanks to Jennifer Smith at iRobot for the modifications to the MIDI system and the LED output drivers.

[2] Only an engineer would call blinking lights "dramatic"…

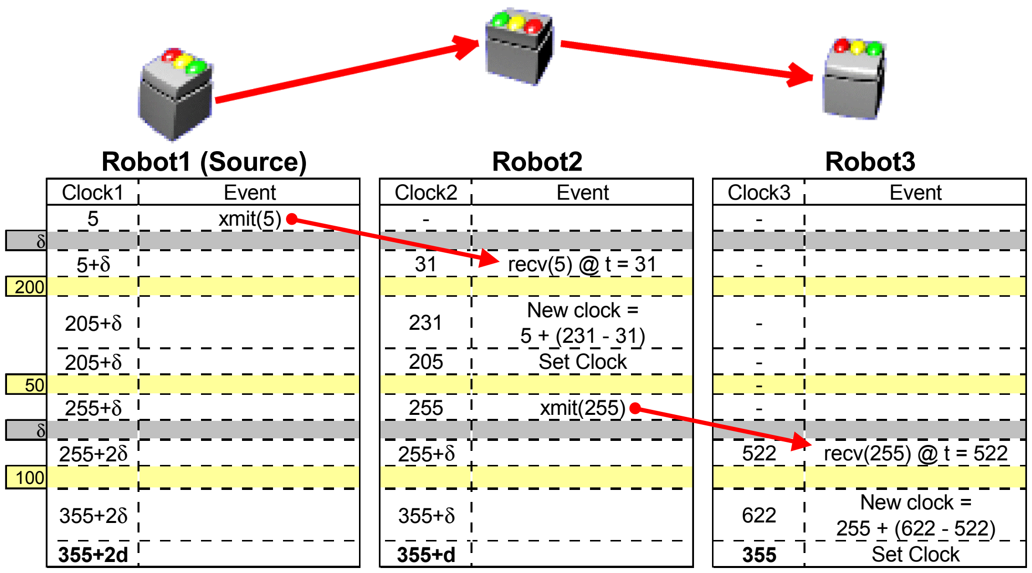
positions of neighboring robots. The temporal sync information is also transmitted at this time. Each temporal sync communications packet contains the following information:

- sourceID
- senderID
- time of transmit
- time of reception

When a non-source robot receives a temporal sync packet from any neighbor, it is compared to its best sync packet received during its current ISIS neighbor cycle If the new packet is from a robot that is closer to the source that your current packet, then keep it and discard the current one. All packets are discarded at the end of the ISIS neighbor cycle. This prevents robots from keeping stale data.

If the robot has received a valid sync packet over the last ISIS neighbor cycle, it adjusts it's own clock according to the information in the packet. For example, if this robot is one hop from the source (i.e. it can communicate directly with the source), it's current time is 231, and the packet information is:

time of transmit = 5
time of reception = 31



	Robot1 (Source)		Robot2		Robot3	
	Clock1	Event	Clock2	Event	Clock3	Event
	5	xmit(5)	-		-	
δ						
	$5+\delta$		31	recv(5) @ t = 31	-	
200						
	$205+\delta$		231	New clock = 5 + (231 - 31)	-	
	$205+\delta$		205	Set Clock	-	
50					-	
	$255+\delta$		255	xmit(255)	-	
δ						
	$255+2\delta$		$255+\delta$		522	recv(255) @ t = 522
100						
	$355+2\delta$		$355+\delta$		622	New clock = 255 + (622 - 522)
	355+2d		**355+d**		**355**	Set Clock

Figure 3: Temporal sync example. Each column represents a different robot. The robot on the left is the temporal source. ISIS communications packets are represented by the red arrows. Temporal synchronization flows from left to right. The gray rows represent the error for each transmit cycle. Yellow rows represent arbitrary time between packet reception and packet processing.

Then it would adjust its clock from 231 to 5 + (231 – 31) = 205. This will make it's clock almost equal to that of the transmitting robot, with some potential errors from interrupt latency and processing lag. Figure 3 presents an example that shows how error creeps into the system. This error is analyzed in detail in the section below.

The time between synchronization events can vary from once per song to every downbeat. Since the robots all use crystal oscillators and the music selections are short, the skew across one song will be imperceptible to the human ear. Therefore, after the robots are synchronized and the song is started on the same downbeat, the robots can rely on their internal clocks to stay coordinated for the remainder of the song. While simple, this approach does not allow for synchronization changes or long-term operation past the limits of oscillator drift.

An alternative is to sync at regular intervals. A real orchestra syncs with each downbeat, which requires extensive local communication and musical interpretation (How to you explain a "downbeat" to a robot?) We implement a simpler version of this, with sync packets being transmitted from the temporal source every 250ms.

Errors and Future Work

A small error δ is introduced at each communication hop. Looking closely, we see that:

$$\delta = c_1 + c_2 + E[t_i] + c_3$$

where:

$c_1 =$ The time between the clock read and the IR packet xmit

$c_2 =$ Packet transmit time

$E[t_i] =$ The interrupt latency

$c_3 =$ The time between interrupt start and time stamp read

All the c_i's can be measured by counting instructions and measuring packet transmit time. The interrupt latency is a function of operating system design. The SwarmOS running the ThreadX Real-Time kernel locks out interrupts for about 6-10 μs, measured with a two-thread text application running under controlled circumstances (no external interrupts, no serial streams, etc..)

This error accumulates as a function of the diameter of the graph. This will be true in any sync system that relies on a single source and local communications relaying to share

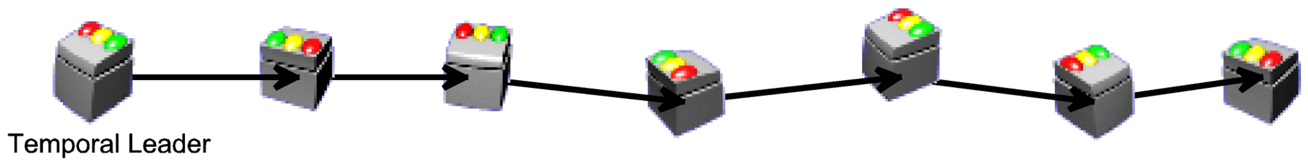


Figure 4:This is the worst case network for temporal synchronization. The sync signal must travel through n-1 robots before it reaches the last robot. Error is accumulated with each hop.

information. Figure 4 shows the worst case topology, where all the robots are arranged in a line. The temporal error across a network like this can be as large as $\delta(n-1) \approx \delta n$. where n is the number of robots in the network. However, since the signs of the error accumulated at each hop are random, the mean of the error can be made to be close to zero, but the standard deviation will grow as a function of $\delta\sqrt{n}$ [Grachaw 2002]. For a large network, this can become unacceptable. With these measurements and bounds, we can zero out most of the propagation error. Fortunately, the average diameter of the Swarm orchestra graph is about three, so this correction was not needed in the current implementation.

If the FPGA that is responsible for encoding and decoding the ISIS packets also time stamps them, this would eliminate the probabilistic portion of the error, the interrupt latency. Additionally, all of the constants could be explicitly measured from the logic synthesis. The error now can be made to be smaller that one system clock tick. In a 40mz SwarmBot, this is 25ns, which is absurdly small, and should suffice for almost all application. Even in groups of thousands of robots, this is still only 25μs in the worst case configuration

5. Distributed Instrumentation

Now that all the robots are playing the same song at the same time, the next step is to get them to use different instruments. I will impose the additional constraints that the swarm should divide evenly amongst the various instruments needed for a song, and robots playing the same instrument should be physically near each other (sax section, drum section, etc…) We will use the following variables in this section:

n = number of robots
i = number of instruments (tracks in the MIDI file)

Initial Approaches

The pheromone gradient system would allow us to pick i leaders easily with lateral inhibition. However, once these leaderBots are selected, it becomes difficult to select an equal number of minionBots in each group. If the minionBots pick the leaderBot that is closest in hop counts or space, this could result in an unequal clump size. In order for the algorithm to vary the clump size, it would have to be able to count the number of robots that are in each clump, or measure some other parameter that is correlated to the number of robots in each clump. This would allow the leaderBots to recruit or release minionBots using some kind of distributed feedback loop (maybe competition?). This could result in all the groups becoming similarly sized, but the two requisite pieces, a distributed counting algorithm and a competition-based recruitment algorithm, were missing from the Swarm's behavior repertoire at the time of implementation.

A behavior that did exist was the ability to tessellate the swarm in the presence of a several leader gradients. Each minionBot pledges allegiance to the leaderBot that it is closest to in terms of hop counts, which results in a Voroni tessellation of the swarm. If the leaders could carefully arrange themselves, it could be possible to divide the swarm evenly in this manner.

This is similar to K-Means clustering [Hastie 01], in which the center of a cluster is iterativly moved closer to the center of mass of the data points [ref]. The missing algorithmic piece for this approach is a distributed computation of the center of mass of each region, and the ability to communicate this to the leaders so they can convert it to spatial coordinates (or gradients (or some other kind of simple taxis)) and move towards it. A refinement of this would be to move the leaders virtually by sending communications messages from robot to robot, but this requires "conservation of leaders" and leads to an infinite handshake, which is a form of the Byzantine Consensus problem, which is provably intractable [Lynch 1996]. Not good.

Solution: Divide-n-Clump

The Divide and clump algorithm takes advantage of the fact that all the robots are mobile, not just the leaders.

```
Divide-n-Clump
   Pick an instrument uniformly at random
   Become a source of a gradient for that instrument
      Gradient inhibition is based on robotID, sources with lower IDs
      dominate.  These robots are the leaderBots.
   If your instrument gradient is being inhibited, move up the gradient
      towards lowest ID for their instrument.  When you get within a
      predetermined range of the source, stop and become clumped.
```

This simple algorithm generates a large mess that eventually settles down into clumps of robots more or less spatially grouped by instrument, as shown in Figure 5. Since all the robots uniformly pick an instrument at the beginning of the execution, the sizes of the groups are fairly equal.

"Spatial Simulated Annealing"

Settling time is a problem with the algorithm as described above. The robots jostle around quite a bit, bumping into each other as they look for their clumps. Some amount of this is good, as random noise helps get unclumped robots into their clumps, but it can also dislodge other clumped robots. This process was damped by using ideas similar to simulated annealing. The robots reduce their speed as they get closer to their clump leader. Once they are clumped, they reduce their speed even further. These relations are shown in the equation below

$$v = (v_{min} + r\alpha)\beta$$

where
$v =$ behavior velocity
$v_{min} =$ minimum velocity
$r =$ range to clump source
$\alpha =$ speed to range ratio
$\beta =$ clumped slow down factor (1 if unclumped)

This reduction in speed helps clumped robots stay put as they are bumped by unclumped robots, yet still lets the swarm move and flow to let unclumped robots reach their destinations. The difference between this and simulated annealing is the use of spatial location instead of time as the input to the energy (temperature) function. This lets individual robots determine how much speed to use based only on their immediate sensor reading of their location., which removes explicit state that must be shared globally, i.e. a global timer, and allows the algorithm to be robust in the face of dramatic topology changes (e.g. when the user moves 1/3 of the robots away from his feet as he is trying to write this paper).

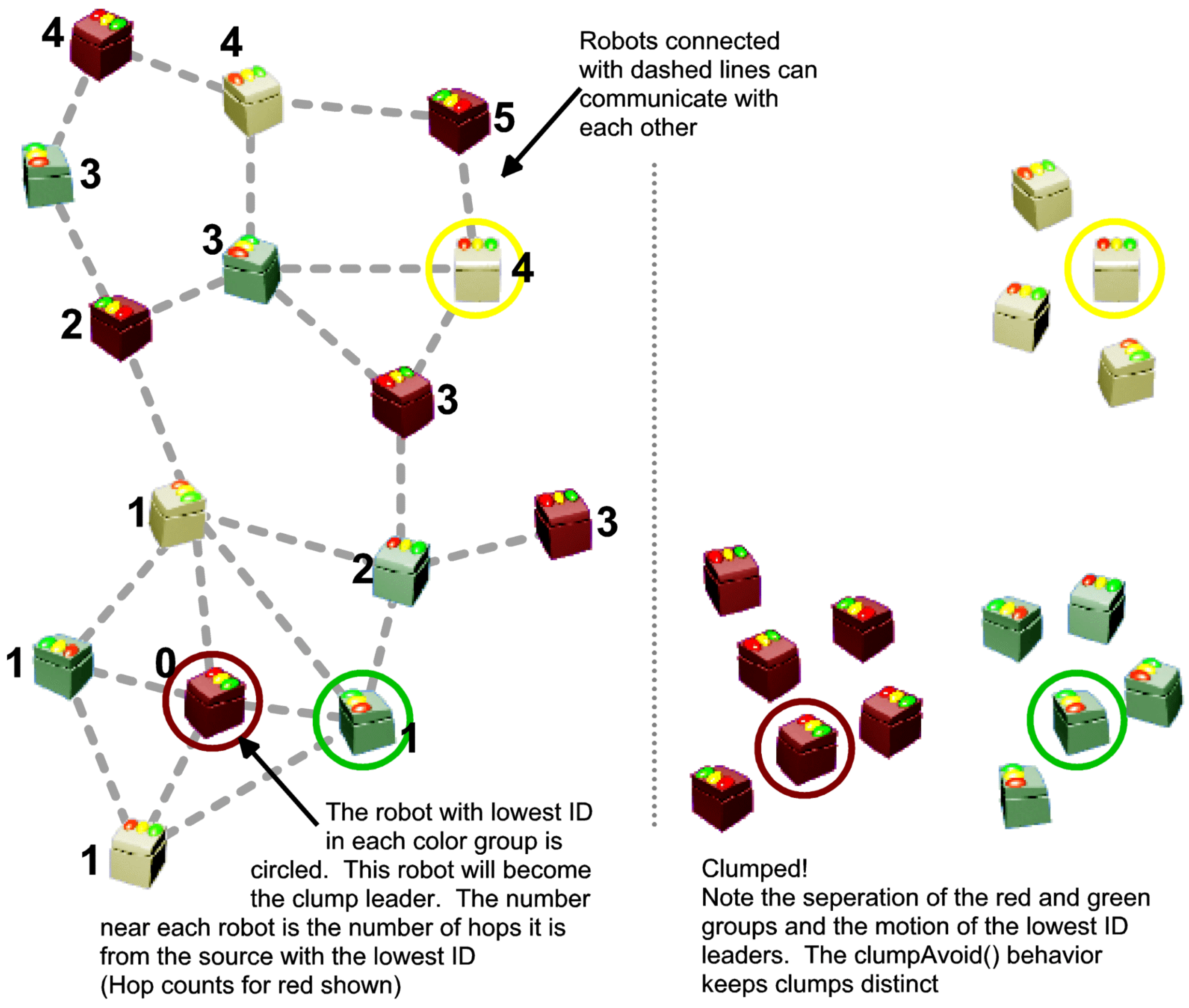

4
4
5
Robots connected with dashed lines can communicate with each other
3
3
4
2
3
1
3
2
1
0
1
1
The robot with lowest ID in each color group is circled. This robot will become the clump leader. The number near each robot is the number of hops it is from the source with the lowest ID (Hop counts for red shown)
Clumped!
Note the seperation of the red and green groups and the motion of the lowest ID leaders. The clumpAvoid() behavior keeps clumps distinct

Figure 5: The Divide and Clump Algorithm generates a spatial division of labor. Robots select their group (red, yellow, or green in the figure above) at random. Each robot then becomes a source for a pheromone gradient corresponding to their group. This gradient is inhibitory, such that gradients from robots with lower IDs override those from others. Eventually, the robot from each group with the lowest ID is the only remaining active gradient source, and all the other robots follow the gradient towards it, producing clumps. These clumps are made distinct by the **`avoidClumps()`** behavior, which moves clumped robots away from each other. Settling time is reduced with a "simulated annealing" behavior.

Separating Clumps

Once the robots are clumped, a **`clumpAvoid()`** behavior separates clumps from each other. Clumped robots avoids any other clumped robots that are part of a different group. Unclumped robots are not avoided, as their presence is likely transitory and does not warrant an avoidance reaction. The avoidance behavior is performed slowly relative to the overall motion of the robots to allow the clumps to be maintained. This was simple to implement, and is surprisingly effective at separating clumps.

Errors and Future Work

The main problem with the algorithm right now is settling time. The need to physically move a large portion of the robots around the network simultaneously causes much chaos. The more elegant solutions of only moving the leaders, or leaving all the robots stationary and moving the clumps virtually with communications should offer large improvements. Also, selecting clumps initially (i.e. once) does not let the swarm dynamically adjust the clump size. Being able to count the robots in each clump and then recruit robots if your current clump is small is needed for robustness.

6. Conclusion

As of this writing, the Swarm Orchestra can clump, sync, and play MIDI files with ten parts. The overall effect is very endearing, especially with the lights flashing as different parts come in and out of the song. The two distributed algorithms presented, temporal sync and can be used as building blocks for other applications.

7. References

Coore, Daniel N., "Botanical Computing" Ph.D. Dissertation, MIT, 1999

Grochow, Joshua [personal communication]

Hastie, Trevor, Robert Tibshirani, Jerome Friedman, "The Elements of Statistical Learning", Springer-Verlag, New York, 2001

Lynch, Nancy, "Distributed Algorithms", Morgan Kaufmann Publishers, San Francisco, CA 94104, 1996

McLurkin, James "Algorithms for Distributed Sensor Networks", Master's Thesis, U.C. Berkeley, 1999

Mirollo, Renato E., Steven H. Strogatz, "Synchronization of Pulse-Coupled Biological Oscillators", Society for Industrual and Applied Mathmatics, Vol. 50, No. 6, pp. 1645-1662, December, 1990

Schneider Fontan, Miguel, Maja J. Mataric, "A Study of Territoriality: The Role of Critical Mass in Adaptive Task Division"