Anton Ridgway

Dec. 19, 2014

**Introduction**

One of the more unusual applications for cyber-physical systems research is the creation of music ensembles that are, in part or in whole, composed of machine performers. Generally, these are systems where each instrument is controlled as an independent component, with a centralized “conductor” node used for synchronization. When human performers are involved, a number of different sensors can be used to collect readings and help keep the computer-controlled instruments playing in-sync with them. The goal of such projects has included the investigation of creating and synchronizing musical robots, as well as the development of the systems to connect them, and the higher-level exploration of the new possibilities created by such systems.

Naturally, there are a wide variety of challenges that come along with creating such a system.

* Drivers must be developed to control the actuators or synthesizers for each instrument.
* Researchers must manage the logistics of what musical sequences are assigned to each node, and how those sequences should be interpreted.
* An overarching vision is needed to drive the development of the system to meet the needs of any given musical piece.

The best-known computer-enabled orchestra, The KarmetiK Machine Orchestra from the California Institute of the Arts, primarily focuses its research efforts on the technical aspects of driving the robots, and on creating new forms of human-computer musical interactions. By contrast, my project lies in a niche between these low-level and high-level concerns. My interest was to develop a system of interchangeable instruments that could be able to distribute the different components of a musical score (for instance, the soprano, alto, tenor, and bass voices in a four-part chorale-style piece) automatically among themselves. More specifically, I investigated the feasibility of implementing distributed algorithms in embedded agents to automate part of the cyber-physical systems development process.

Such a system would primarily be useful to rapidly iterate design decisions for these ensembles; given a group of computer-controlled instruments, it would be useful to have a protocol in place for the instruments to communicate and distribute the different parts of the score among themselves according to some specified proportion and their own capabilities, rather than needing to reprogram each component each time. Additionally, such a system would be more robust than a hand-programmed system, since a simple re-application of the protocol would be sufficient to reorganize in the event of component failure.

Clearly, in creating such an automation, it’s crucial that some generalized design be implemented across all instruments, which at the same time accommodates the need for some specific intelligence about how the ensemble parts are best distributed across the instruments. In my case, I supplied each instrument with a capability profile; these profiles represent factors such as the range of pitches it can reach, for example, or the strength of its timbre—melodic instruments like the piano are often given dramatically different tasks in an orchestra than percussion instruments like the drums would be.

An additional concern is agent communication and message processing. Agents are a natural way to think about developing cyber-physical systems in general, and they seem a good fit to help provide a uniform interface for the different components. However, in doing so we also need to address the challenges that can arise in implementing agents in resource-restricted embedded systems. In this project, I was able to implement a simple three-instrument system that makes use of a basic messaging scheme to carry out a backtracking protocol. These initial tests showed that a basic agent-based system could easily be fit on embedded devices and run with sufficient operating speed.

**Literature Survey**

Computer-driven orchestras, while interesting, have not been as thoroughly studied in existing literature as many other topics. With this in mind, my background material survey examines both specialized efforts in this specific area, as well as more general works that have to do with agents and abstraction in a cyber-physical context.

The KarmetiK Machine Orchestra project involves a mixture of human and computer performers. A star network is employed to connect the various components, complete with a central server node to provide low-latency clock pulses to client instruments (Vallis 67) and accommodate musical pieces with changing tempo (66). An XML-based API is also used to allow the server access to instrument capabilities, as well as to notify it if it encounters a malfunction (68), and note data is directly transmitted to the instruments from a centralized source, using the OSC and USB-MIDI protocols (68).

It is worth noting that for KarmetiK, human performers’ interactions with the system are a primary focus, since the computer-controlled components allowed the developers of the project to investigate new modes of performer-instrument interaction. This included a system which created a many-to-many relationship between performers and instruments—so that each performer affected many instruments, but each instrument’s actions were driven by input from multiple performers—and another which allowed the performers to interact with the machine instruments’ mathematically-driven rhythms through a Java-based interface (69-70).

My own project takes some cues from this one. Synchronized pulse information is provided by a central “master” node, and a uniform interface is used to encapsulate capability information. However, I did not account for human input during the performance, and I considered tempo to remain fixed, eliminating many synchronization concerns. In a full-scale system, a synchronization technique like that used by the KarmetiK Orchestra could be implemented, and some gesture-based input could even be used to adjust the tempo to keep an appropriate pace with human performers. Because embedded processors do not typically have any sense of absolute time, a centralized pulse of some kind seems necessary to reduce the impact of varying clock speeds in any case.

Messaging in my project has also been simplified; rather than a full MIDI protocol or something on that scale, a minimal messaging system appropriate for the limited inputs and outputs of embedded systems has been developed in tandem with a coordination protocol.

An analysis of HCI contexts for music by Wanderley and Orio (69) presented a classification of seven possible levels of musical interaction:

1. Note-level control, affecting pitch, loudness, and timbre of an instrument
2. Score-level control, affecting a previously defined musical sequence (ie, affecting composed music by altering notes to be played, or their order)
3. Sound processing control, applying filtering effects to digital audio
4. Traditional HCI contexts, including typical gesture controls such as drag-and-drop and scrubbing
5. Multimedia installation interaction, including broad gestural interactions from one or many people
6. Dance/music interface interaction, focused primarily on chorographic music control
7. Computer game control, focused on affecting music through games

Vallis et al. note that in the context of computer-controlled orchestras, the first three categories are generally the most relevant, while the latter four have more to do with more interactive listener-performer experiences. As long as pre-composed musical score is known to the embedded components in advance, note-level, score-level, and sound-processing-level improvisation is possible for the instrument computers (64).

In my own project, I do not make use of improvisational techniques, but it is true that this categorization is useful to help break down musical control and understand musical structure in a computer-orchestra scenario. In my model, score-level control is reserved for the master node only, while note-level and sound-processing-level improvisation would fall easily within the purview of the instrument agents, if their implementation became a project goal.

In a case study on water distribution networks by Lin (“Modeling”), a semantic service layer was implemented as a method to provide agent abstraction in a cyber-physical system. In particular, they specify three components which help to accomplish this (15-17). They provide an ontology layer, which provides a formal specification of the environment, a semantic service layer, which uses the ontology to interpret incoming data, and an agent layer, which carries out the actual decision-making process based on the semantic data (14). The goal of this semantic processing is to consolidate incoming data and reduce redundancy, provide some preprocessing and data abstraction, and give users semantic query capabilities (16). In my own project, I didn’t necessarily have the processing time or program space for a distinct semantic interpretation layer, but I did take the time to develop a data model that is used to categorize stored data, and I stored the information in a way that provides easy access for agents, and can be easily “queried” by developers by inserting serial trace commands on the Arduino devices used for implementation.

This case study was expanded upon in another work (“Agent-based”), where the authors focused on data heterogeneity issues, in context of varying data formats and scale in a wireless sensor network. They observe that data heterogeneity complicates decision-making algorithms, especially with respect to data storage structures, device configuration, and physical infrastructure data. Dealing with many different types of data also reduces implementation portability between different systems, and exacerbates issues with data volume or accuracy (93). To combat this, they use the same semantic processing layers described before, but add support for imprecise queries; that is, agents in the agent layer can query the knowledge representation database using vague access terms, and data can be categorized within a certain semantic “thesaurus” distance of the vague terms is returned (99).

This algorithm in particular is too high-level for our embedded systems, but the issues it raises do hold true for heterogeneous systems in general. Our machine orchestra implementation uses a tightly controlled messaging structure to ensure that data heterogeneity cannot become an issue, and a uniform structure is provided to contain the naturally heterogeneous configuration and physical infrastructure information provided by the master node.

Finally, a brief paper by Reijers discusses the development of middleware in wireless sensor networks, to enhance code portability by reducing the strong coupling that typically ties low-level sensor agent code to the specifics of particular networks. Though their study is incomplete, they discuss the issues associated with developing CPS code that is minimally constrained by deployment—what sensors are in the systems, or where they are deployed (1). Their goal is for a master node to be able to reconfigure the CPS configuration centrally, according to needs specified by users, detected failures, or optimization goals (2). To aid in developing a higher-level system to develop on top of existing CPS structures, they develop an agent classification, separating the agents equipped with a common communication protocol from those that are not, and require that each agent is able to supply a capabilities profile (2). Further, they lay out the need for three independent frameworks: a sensor profile framework allows the master to deal with heterogeneous agent types and logical abstraction, an application policy framework that allows for the specification of execution objectives, and a real-time decision framework that enhances system flexibility (2).

Like my own project, this approach specifies a common communications protocol, and maintains capabilities profiles. My project also makes use of a similar topology to Reijers’; a master node is also used to configure the system, and though it only provides initial broadcasted instructions rather than ongoing control, trusting agents to communicate among themselves to distribute tasks, it would allow for an easily implemented user interface to configure the system.

**Agent Considerations**

Part of our goal in this project is to encapsulate implementation details of CPS components using an agent model; but in doing this it’s important to first understand what considerations drive agent structure and development in general. The accepted definition of an agent is actually fairly broad—strictly speaking, agents are defined as programs that make actions based on their perceptions, and possess some configuration of sensors and actuators (Russell 46). This corresponds well with the structure of typical cyber-physical components, which also includes physical sensors and actuators as input and output. Accordingly, in designing the agent for this project, I considered information supplied from the master node to represent perceptions, while the tones that can be produced through the musical hardware are the actions. Actual sensors were not used in my case, but they would be important to a system like the KarmetiK orchestra that incorporates human performers as well.

In designing agent systems, we should also pay attention to how we can characterize the agents’ environment. In Russell’s text, he specifies various qualities we can look for—observability, determinism, episodicity, dynamism, discreteness, and knowability (43-44).

**Observability:** Multi-agent systems are not fully observable in general, since each agent typically has its own restricted view rather than a full perspective on everything. This holds true in our particular case, where agents only have access to the information sent them by the master node and other agents, so we classify the environment as only partially observable.

**Determinism:** We do assume that the environment is fully deterministic, so that when we try to play a tone we know it will be played correctly. At the same time, a more realistic agent model in a full-scale project might also need to account for sources of “random” unexpected error, such as connection loss between parts, or actuator failures, and compensate accordingly.

**Episodicity:** We can also say that the environment in our case is episodic, since each action the agent makes need not be informed by its previous actions. However, if instrument improvisation were implemented (as the KarmetiK researchers considered), this episodicity would be lost, since we would need to keep track of what changes we had made to the initial musical sequence to be able to continue correctly.

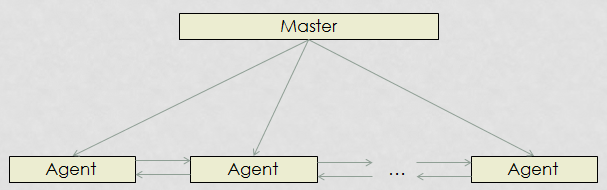
**Dynamism:** Since music is normally tightly time-constrained, we also say that the environment is dynamic—if an agent is unable to make a decision within the time threshold, it will be equivalent to choosing to do nothing. To account for this in my system, I made use of a song whose tempo allows sufficient time for agents to complete any computations. In a full-scale system, however, we would need to account for this failure case by either choosing a “best-guess” when we can no longer afford to wait, or by skipping forward to catch up with the other instruments.

**Discreteness:** We can say that the agent environment is discrete, since we restrict the possible actions of the agent (i.e., the number of tones it can be playing) to those within the traditional twelve-tone system. This brings with it potential simplifications of agent messaging, since there are a finite number of known agent actions we will need to deal with.

**Knowability:** Although agents have varying logic according to their capabilities, we expect each of them to output audio according to the same structure, so we consider the environment to be known (i.e., we are aware of the rules by which it changes). This consideration underpins our ability to take relatively larger amounts of time between tones in a dynamic environment—we know how we expect the environment to have changed in that time.

**Agent Design**

The software design of my agents was fairly straightforward, including a simple data model and a three-stage distributed algorithm. As mentioned before, I structured my system to make use of a single master node that provides initialization information and a constant synchronization pulse.



During the first stage of the algorithm, initialization, an agent listens for broadcast messages from the master node, and interprets the messages in order to fill out its view (represented by the internal data model). These messages contain control information from the user of the system, including how many agents are to be used, and what proportion of the agents should be assigned to each possible role. Once the expected number of messages is received, the agent moves to the second stage, coordination, where it listens for messages from other agents to fill out its view of their state, and finally decides on which role it will take. Five agent roles are allowed in this implementation—soprano, bass, alto, tenor, and percussion. Finally, once all agents are satisfied with their chosen roles, they move to the last stage, performance, where they play the musical score associated with each role in-sync. Because Arduino input and output in the LittleBits system I worked with is somewhat limited, I designed the system to keep messaging to a minimum.

Initialization and performance are fairly simple, sequential activities, so most of the sophistication in this design comes in the coordination phase. Coordination is structured as a distributed backtracking algorithm, where agents are divided into a hierarchy by ID number. The initial “turn” to choose a role is given to the agent with ID 0, and each agent is responsible to choose a role on its turn, then broadcast its decision to the other agents and pass control on to the next-highest priority agent. Agents decide on a role based on their own preferences table (which encapsulates its capabilities, eliminating impossible roles and a designating a simple ranking of which roles it prefers), while also eliminating some roles outright, depending what has been rejected by other agents over the course of the backtrack, and what is disallowed by the ideal ensemble composition from the master node. If no role is acceptable, it notifies its superior that its current choice is unacceptable, and passes control back up. These rejection messages are stored in a table by the receiving agent, and used to prevent any repetition of states in the backtrack. This process allows each agent to bring its preferences and capabilities to bear on the decisions it makes, without needing to transmit them to any other agent. If all agents are able to find a satisfactory assignment, a confirmation message notifies all agents to begin performance on the next pulse.

The data model developed to support this algorithm contains three main types of data. It stores environment information, such as how many agents were allowed, how agents are ideally to be distributed among the different positions, and what score the agents are meant to play for each part. Agent view information was also stored, which helps both to inform its decision on which role it would take (depending on the roles chosen by the other agents, its own preferences, and roles that has previously been rejected by the backtracking process), and to track the overall state of the algorithm (i.e., who has control) and allow all agents to move to the performance stage at once. Stored timing info also helped to demarcate how many pulses should be devoted to each note that was played, as well as brief pauses between notes.

Pseudo-code for the Algorithm:

If pulse value changes,

If we're in the initialization stage,

If another message is transmitted, store it in the correct location

as defined by a pre-defined order.

If that was the last message we expect, move to coordination stage.

Else if we're in the coordination stage,

If we need to broadcast a message,

Either wait (on either side of the broadcast pulse)

Or transmit the message.

If we just transmitted a confirmation, skip the extra wait pulse

and move to performance stage.

Else

Read in scaled input data.

If there's a new message, store it.

If we did get a message

If it's a higher-priority agent's turn

If it's a CHOICE message

Update the view.

Increment the current-turn counter.

If it's a REJECT message

Decrement the current-turn counter.

Update the view.

Else if it's a lower-priority agent's turn

If it's a CHOICE message,

Update my view.

Increment the current-turn counter.

Else if it's a REJECT message,

If it's from the agent directly beneath me in the hierarchy,

store my current role in the rejected roles table.

Decrement the current-turn counter.

Else if it's a CONFIRM message,

Move to the performance stage.

If it's my turn

Choose the best role I can (given constraints) and update my view.

If I couldn't choose any role,

Decrement the current-turn counter.{

Get ready to broadcast a REJECT message.

Clear out my rejected-value table.

Else,

If I'm the lowest agent in the hierarchy,

Get ready to broadcast a CONFIRM message.

Else,

Get ready to broadcast a CHOICE message.

Else if we're in the performance stage,

If we're percussion, output according to percussion timing.

Else, output according to position in my score and melodic timing.

Data Model:

Environment Info

* # agents
* Ideal composition table
* composition tolerance factor
* musical score table

Agent View

* my ID number
* preferences table
* agent-view table
* rejections table
* ID of current-turn agent
* most recent message transmitted (for querying/debugging)
* current physical inputs
* current algorithm stage

Timing Info

* # play-pulses
* # wait-pulses

Four message types were also designed, each using a single value to denote both its contents and its message type. Initialization messages are simply intended to transmit data in sequence and be recognized by context. Choice, Reject, and Confirm messages were used for coordination, where Choice indicates when an agent has chosen a role (which one can be determined by the value transmitted), Reject indicates that an agent cannot make a satisfactory choice, and Confirm indicates that every agent has been able to make a satisfactory choice.

Message Types:

**INITIALIZATION**

-Next expected value

**CHOICE**

-Chosen position value (from 20, 50, 80, 110, 140)

**REJECT**

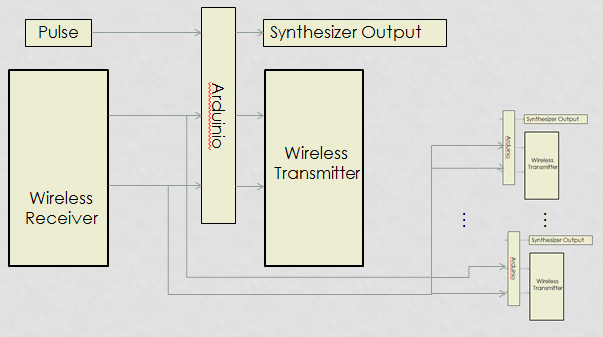
-170

**CONFIRM**

-200

**Hardware Implementation**

The hardware design of my system was fairly straightforward. I utilized a pulse component (taking the place of the master node) to keep the agents in-sync, and Arduinos with synthesizers were used to represent the instruments. Due to the limited number of components available, it was necessary that every message be broadcast to all agents; this drove my decision to build the use of broadcast messages into the structure of my coordination algorithm.



Though a master node for system initialization was part of the initial design, the limitations on part availability mentioned before made this infeasible, so I needed to hard-code all the information that would normally be provided by this node at the start of the program, and begin all agents at the coordination stage.

**Demo Results**

For my demo scenario, I used three agents. Agent 0 was able to perform both soprano and bass, but preferred soprano; Agent 1 could only perform soprano, and Agent 2 could only perform percussion. In the ensemble proportions table, I specified that only one agent each would be used for soprano, bass, and percussion. This layout allowed me to demonstrate both melodic and percussion roles (which are handled separately during performance), while also allowing for some backtracking to occur during coordination (since Agent 0 will initially try to claim soprano, and need to backtrack to choose bass). As it turns out, the pulse speed did need to be increased between the coordination and performance phases, but this could be handled in a full-scale system by allowing the master node to listen for the confirmation message and adjust it automatically. Alternately, the number of pulses used for each note during performance could be decreased to avoid needing to change the pulse between phases, or the delay for each pass of the loop (currently 3ms) could be adjusted.

I used a fourth Arduino with a wireless receiver to listen in on transmitted messages (by forwarding inputs to a serial output whenever they changed significantly) to ensure they were working correctly. This monitoring process generated the following output, commented after the fact. (Note that this listener node was not restricted by the same pulse as the other agents, so the rising and falling edges of some messages are visible.)

[0,-1,0]

[0,21,0] //20 --> Soprano claimed by Agent 0

[0,5,0]

[0,0,0]

[0,134,0]

[0,162,0]

[0,169,0] //170 --> Agent 1 rejects this arrangement

[0,146,0]

[0,-1,0]

[0,119,0] //110 --> Agent 0 takes bass

[0,107,0]

[0,4,0]

[0,0,0]

[0,18,0] //20 --> Agent 1 takes soprano

[0,1,0]

[0,15,0]

[0,214,0]

[0,193,0] //200 --> Agent 2 picks percussion and sends CONFIRM

[0,196,0]

[0,26,0]

[0,0,0]

It is worth noting that some difficulty was encountered in developing the demo because the Arduino inputs were read in on the scale [0,1023], while the outputs were transmitted on the scale [0, 255]. Additionally, around 5 units of constant noise (on the former scale) were constantly received from the wireless receiver. I needed to account for both of these factors to make hardware-based messaging perform as expected.

**Conclusion**

Initial tests showed with a nearly full-scale program showed that the Arduino-based LittleBits system chosen for these tests provided sufficient flexibility to implement an agent, considering both memory and speed. I was able to keep the program to a manageable size (only about 10kb of 28kb onboard memory was used), while serial output seemed to show that the processor was able to keep pace sufficiently with the speed of the pulse component I used for synchronization.

Future work might involve adding support for more types of components, or investigating possibilities of using non-broadcast messaging to simplify the protocol. Certainly it would be important to eventually implement the master node, but it would also be nice to be able to provide an interface between it and a PC to allow a user to designate the parameters to be broadcast during initialization.

**Bibliography**

Jing Lin; Sedigh, S.; Hurson, A.R., "An Agent-Based Approach to Reconciling Data Heterogeneity in Cyber-Physical Systems," Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on , vol., no., pp.93,103, 16-20 May 2011  
doi: 10.1109/IPDPS.2011.130 <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6008824&isnumber=6008799>>

Jing Lin; Sedigh, S.; Miller, A., "Modeling Cyber-Physical Systems with Semantic Agents," Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual , vol., no., pp.13,18, 19-23 July 2010  
doi: 10.1109/COMPSACW.2010.13 <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5614900&isnumber=5614033>>

Reijers, N.; Yu-Chung Wang; Chi-Sheng Shih; Hsu, J.Y.; Kwei-Jay Lin, "Building intelligent middleware for large scale CPS systems," Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on , vol., no., pp.1,4, 12-14 Dec. 2011  
doi: 10.1109/SOCA.2011.6166238 <<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6166238&isnumber=6166194>>

Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall. 2009.

Vallis, Owen; Diakopoulos, Dimitri; Hochenbaum, Jordan; and Kapur, Ajay. “Building on the Foundations of Network Music: Exploring interaction contexts and shared robotic instruments,” Organised Sound, 17, pp 62-72. 2012. doi:10.1017/S1355771811000525. <<http://dx.doi.org/10.1017/S1355771811000525>>

Wanderley, Marcelo Mortensen; Orio, Nicola. “Evaluation of Input Devices for Musical Expression: Borrowing Tools from HCI,” Comput. Music J. 26, 3 (September 2002), 62-76. DOI=10.1162/014892602320582981 <<http://dx.doi.org/10.1162/014892602320582981>>

**Project Arduino Code:**

//Provides appropriate NOTE\_ constants; from Arduino example projects.

#include "pitches.h"

//Arduino Pins

const int INPUT1 = A0;

const int INPUT2 = A1;

const int OUTPUT1 = 5;

const int OUTPUT2 = 9;

const int PULSE\_IN = 0;

const int MUSIC\_OUT = 1;

//Mode constants

const int S\_INITIALIZATION = 1;

const int S\_COORDINATION = 2;

const int S\_PERFORMANCE = 3;

//Message Constants

const int M\_REJECT = 170;

const int M\_CONFIRM = 200;

const int MESSAGE\_TOLERANCE = 15;

const int POSITION\_VALS[] = {20,50,80,110,140};

const int POSITION\_TYPES = 5;

//Musical Role Constants

const int P\_NONE = 0;

const int P\_SOPRANO = 1;

const int P\_ALTO = 2;

const int P\_TENOR = 3;

const int P\_BASS = 4;

const int P\_PERCUSSION = 5;

//Performance Mode Constants

const int PMODE\_PLAYING = 1;

const int PMODE\_WAITING = 2;

//Environment Info

int numAgents = 3;

//Proportion information to determine how we choose a role. Here, we want one agent each for

//soprano, bass, and percussion.

int propTable[] = {1,0,0,1,1};

int propThreshold = 0; //In the demo, no tolerance is allowed in choosing roles.

// Notes in the score; used in performance mode. For this demo, alto score is

// the same as soprano and tenor is the same as bass.

int sopranoScore[] = {

NOTE\_E4, NOTE\_D4, NOTE\_C4, NOTE\_B3, NOTE\_A3, NOTE\_G3, NOTE\_A3, NOTE\_B3};

int bassScore[] = {

NOTE\_C4, NOTE\_G3, NOTE\_A3, NOTE\_E3, NOTE\_F3, NOTE\_C3, NOTE\_F3, NOTE\_G3};

int percussionWeights[] = {NOTE\_D3,NOTE\_D2,NOTE\_G2,NOTE\_D2};

//Timing values used in performance mode.

int tonePulses = 32;

int waitPulses = 8;

int percussionHitPulses = 2;

int percussionWaitPulses = 8;

int performancePulses = 0;

int currentTone = 0;

//A timer is used to skip a pulse before and after broadcasting a message.

//This ensures that we won't read in our own message as a new input.

int broadcastMessageTimer = 0;

const int BROADCAST\_TICKS = 3;

int broadcastMessage[] = {0,0};

//Received message data structures

int thisPulseInput[] = {0, 0};

int mostRecentMessage[] = {0, 0};

const int noiseLevel = 5; //Around 5 units of noise generally present in received messages.

const float inputDivision = 4.0117647058823529411764705882353; //The value of 1023/255

//Some pulses are initially skipped to allow agents lower in the hierarchy to prepare to

//receive the top agent's initial message.

int skipPulses = 3+numAgents-myID;

//Values used to keep backtracking position; these track whose turn it is, and when we should move.

int whoseTurn = 0;

boolean rejectTable[] = {false,false,false,false,false};

int pulseVal = LOW;

//These values may change for each agent in this demo application, so they've been separated out.

int prefTable[] = {0,0,0,0,9}; //Values in range 1-10; 0 means exclude completely

int myID = 2;

int viewTable[] = {P\_NONE,P\_NONE,P\_NONE};

int stage = S\_COORDINATION;

int performanceMode = PMODE\_PLAYING;

void setup()

{

pinMode(INPUT1, INPUT);

pinMode(INPUT2, INPUT);

pinMode(OUTPUT1, OUTPUT);

pinMode(OUTPUT2, OUTPUT);

pinMode(PULSE\_IN, INPUT);

pinMode(MUSIC\_OUT, OUTPUT);

analogWrite(OUTPUT1,0);

analogWrite(OUTPUT2,0);

analogWrite(MUSIC\_OUT,0);

}

void loop()

{

int newPulseVal = digitalRead(PULSE\_IN);

if(pulseVal != newPulseVal)

{

pulseVal = newPulseVal;

if(skipPulses > 0) skipPulses--;

else

{

if(stage == S\_INITIALIZATION)

{

//Nothing for this implementation... values are provided at initialization.

delay(10);

}

else if(stage == S\_COORDINATION)

{

if(broadcastMessageTimer == 2)

{

if(thisPulseInput[0] <= 5) //Broadcast only if no-one else already is.

{

broadcastMessageTimer--;

analogWrite(OUTPUT1, broadcastMessage[0]);

analogWrite(OUTPUT2, broadcastMessage[1]);

if(broadcastMessage[1] == M\_CONFIRM)

stage = S\_PERFORMANCE; //wait until we've sent it!

}

}

else if(broadcastMessageTimer == 1 || broadcastMessageTimer == 3)

{

analogWrite(OUTPUT1, 0);

analogWrite(OUTPUT2, 0);

broadcastMessageTimer--;

}

else

{

analogWrite(OUTPUT1, 0);

analogWrite(OUTPUT2, 0);

thisPulseInput[0] = analogRead(INPUT1);

thisPulseInput[0] = (thisPulseInput[0]-noiseLevel)/inputDivision;

thisPulseInput[1] = analogRead(INPUT2);

thisPulseInput[1] = (thisPulseInput[1]-noiseLevel)/inputDivision;

if(thisPulseInput[1] > POSITION\_VALS[0] - MESSAGE\_TOLERANCE) //A message

{

mostRecentMessage[0] = thisPulseInput[0];

mostRecentMessage[1] = thisPulseInput[1];

}

if(thisPulseInput[1] > 5) //Got a message (ignore noise)

{

if(whoseTurn < myID) //Higher-priority agent acting

{

//Choice message

if(thisPulseInput[1] > POSITION\_VALS[0]-MESSAGE\_TOLERANCE &&

thisPulseInput[1] < POSITION\_VALS[4]+MESSAGE\_TOLERANCE)

{

viewTable[whoseTurn] = interpretChoiceMessage(thisPulseInput[1]);

whoseTurn++;

}

else if(thisPulseInput[1] > M\_REJECT - MESSAGE\_TOLERANCE &&

thisPulseInput[1] < M\_REJECT + MESSAGE\_TOLERANCE)

{

whoseTurn--;

viewTable[whoseTurn] = P\_NONE;

}

}

else if (whoseTurn > myID) //Lower-priority agent

{

if(thisPulseInput[1] > M\_REJECT - MESSAGE\_TOLERANCE &&

thisPulseInput[1] < M\_REJECT + MESSAGE\_TOLERANCE)

{

if(whoseTurn == myID+1)

rejectTable[viewTable[myID]-1] = true;

whoseTurn--;

}

else if (thisPulseInput[1] > POSITION\_VALS[0] - MESSAGE\_TOLERANCE &&

thisPulseInput[1] < POSITION\_VALS[4] + MESSAGE\_TOLERANCE)

{

viewTable[whoseTurn] = interpretChoiceMessage(thisPulseInput[1]);

whoseTurn++;

}

else if (thisPulseInput[1] > M\_CONFIRM - MESSAGE\_TOLERANCE &&

thisPulseInput[1] < M\_CONFIRM + MESSAGE\_TOLERANCE)

{

stage = S\_PERFORMANCE;

}

}

}

if(whoseTurn == myID)

{

viewTable[myID] = chooseBestValue();

if(viewTable[myID] == P\_NONE) //failure

{

whoseTurn--;

broadcast(M\_REJECT);

for(int i = 0; i < sizeof(rejectTable)/sizeof(int); i++)

rejectTable[i] = false;

}

else //success

{

if(myID == numAgents-1)

{

broadcast(M\_CONFIRM);

}

else

{

broadcast(POSITION\_VALS[0]);

whoseTurn++;

}

}

}

}

}

else //stage == S\_PERFORMANCE

{

analogWrite(OUTPUT1, 0);

analogWrite(OUTPUT2, 0);

if(viewTable[myID] == P\_PERCUSSION)

{

playPercussion();

}

else if(viewTable[myID] != P\_NONE &&

viewTable[myID] != P\_PERCUSSION &&

currentTone < sizeof(sopranoScore)/sizeof(int))

{

playNote();

}

delay(10);

}

}

}

delay(3);

}

void broadcast(int messageType)

{

if(broadcastMessageTimer == 0)

{

broadcastMessageTimer = BROADCAST\_TICKS;

broadcastMessage[0] = myID;

if(messageType == POSITION\_VALS[0])

broadcastMessage[1] = POSITION\_VALS[viewTable[myID]-1];

else if(messageType == M\_CONFIRM)

broadcastMessage[1] = M\_CONFIRM;

else if (messageType == M\_REJECT)

broadcastMessage[1] = M\_REJECT;

}

}

int interpretChoiceMessage(int messageVal)

{

if(messageVal < POSITION\_VALS[0]-MESSAGE\_TOLERANCE || messageVal > POSITION\_VALS[4]+MESSAGE\_TOLERANCE)

return P\_NONE;

else if (messageVal < POSITION\_VALS[1]-MESSAGE\_TOLERANCE)

return P\_SOPRANO;

else if (messageVal < POSITION\_VALS[2]-MESSAGE\_TOLERANCE)

return P\_ALTO;

else if (messageVal < POSITION\_VALS[3]-MESSAGE\_TOLERANCE)

return P\_TENOR;

else if (messageVal < POSITION\_VALS[4]-MESSAGE\_TOLERANCE)

return P\_BASS;

else// if (messageVal <= POSITION\_VALS[4]+MESSAGE\_TOLERANCE)

return P\_PERCUSSION;

}

int chooseBestValue() //Use prefTable, viewTable, rejectTable

{

int spotsTable[] = {propTable[0],propTable[1],propTable[2],propTable[3],propTable[4]};

for(int i = 0; i < myID; i++) spotsTable[viewTable[i]-1] -= 1;

int choiceTable[] = {prefTable[0],prefTable[1],prefTable[2],prefTable[3],prefTable[4]};

int maxIdx = 0;

int maxVal = 0;

for(int i = 0; i < 5; i++)

{

if(spotsTable[i] <= 0 || rejectTable[i] == true)

choiceTable[i] = 0;

else if (choiceTable[i] > maxVal)

{

maxIdx = i;

maxVal = choiceTable[i];

}

}

if(maxVal == 0) return P\_NONE;

else if(maxIdx == 0) return P\_SOPRANO;

else if(maxIdx == 1) return P\_ALTO;

else if(maxIdx == 2) return P\_TENOR;

else if(maxIdx == 3) return P\_BASS;

else if(maxIdx == 4) return P\_PERCUSSION;

else return P\_NONE;

}

void playNote()

{

performancePulses++;

if(performanceMode == PMODE\_PLAYING && performancePulses >= tonePulses)

{

noTone(MUSIC\_OUT);

performanceMode = PMODE\_WAITING;

performancePulses = 0;

currentTone++;

}

else if (performanceMode == PMODE\_WAITING && performancePulses >= waitPulses)

{

int nextNote = 0;

if(viewTable[myID]==P\_SOPRANO)

nextNote = sopranoScore[currentTone];

else if (viewTable[myID]==P\_ALTO) //tones not implemented here

nextNote = sopranoScore[currentTone];

else if (viewTable[myID]==P\_TENOR) //tones not implemented here

nextNote = bassScore[currentTone];

else if (viewTable[myID]==P\_BASS)

nextNote = bassScore[currentTone];

tone(MUSIC\_OUT, nextNote, 10000);

performanceMode = PMODE\_PLAYING;

performancePulses = 0;

}

}

void playPercussion()

{

performancePulses++;

if(performanceMode == PMODE\_PLAYING && performancePulses >= percussionHitPulses)

{

noTone(MUSIC\_OUT);

performanceMode = PMODE\_WAITING;

performancePulses = 0;

currentTone = (currentTone+1)%4;

}

else if (performanceMode == PMODE\_WAITING && performancePulses >= percussionWaitPulses)

{

tone(MUSIC\_OUT, percussionWeights[currentTone], 10000);

performanceMode = PMODE\_PLAYING;

performancePulses = 0;

}

}