Andante:
Composition and Performance with Mobile Musical Agents*

Leo Kazuhiro Ueda† and Fabio Kon
Department of Computer Science
Institute of Mathematics and Statistics – University of São Paulo
http://gsd.ime.usp.br/andante
{lku,kon}@ime.usp.br

Abstract

Across the Centuries, musicians have always had interest in the latest scientific achievements and have used the latest technologies of their time to produce musical material. Since the mid-20th Century, the use of computing technology for music production and analysis has been increasingly common among music researchers and composers. Continuing this trend, more recently, the use of network technologies in the field of Computer Music turned out to be a natural research goal. Our group is investigating the use of mobile agent technology for the creation and performance of music within a distributed computing environment. We believe that this technology has the potential to foster new ways of composing, distributing, and performing music.

This paper describes a prototype implementation of Andante, an open-source infrastructure for the construction of distributed applications for music composition and performance based on mobile musical agents. We also describe two sample applications built using this infrastructure.

1 Introduction

Composers have always looked at contemporary scientific achievements to devise new forms of producing their art. The traditional Western music itself went through changes as new forms of producing sound were being discovered and new instruments were being devised.

Over the past decades, we have witnessed an astonishing development of Computer Science that have led to an intensification of this relationship between Music and Science (Roads 1996). In recent years, the network technologies, especially the Internet, brought us many new possibilities for music making.

In this context, we are interested in discovering how an advanced concept in Computer Science, namely, *mobile agents*, can be applied to introduce new forms of musical composition, distribution, and performance.

The Andante project offers a software infrastructure which allows the construction of distributed applications that use *mobile musical agents* to compose and perform music. Using Andante, programmers can write their own agents in order to build such applications. We are currently working with composers and researchers in writing musical pieces and it is our wish to attract the interest of more people to use this infrastructure for conducting musical experiments.

In (Ueda and Kon 2003), we introduced the Andante project describing preliminary implementations of both the infrastructure and the first sample application. In this paper, we give more details of the infrastructure architecture and describe the latest, more mature implementation, including changes and extensions incorporated in the past months and a new, more sophisticated sample application.

Before that, in Section 2, we describe the mobile agents model in the context of distributed systems in order to define the concept of mobile musical agent in Section 3. Section 4 describes the infrastructure and Section 5 the two sample applications build on top of the infrastructure. In Section 6 we discuss the next steps of our research.

2 Mobile Agents

A *mobile agent* is a computer program that can interrupt its execution on a host, migrate to another host travelling through a network, and resume its execution on the new host (Kotz and Gray 1999). It is an autonomous program in the sense that it can decide itself to migrate and it may react to changes on the host environment.

*Partially supported by a grant from CNPq, Brazil, process number 55.2028/02-9.

†Partially supported by a graduate fellowship from CAPES, Brazil.

This concept, introduced in the mid-90's (Johansen et al. 1994), brought a new paradigm for the construction of distributed and mobile computer systems. In the end of the same decade, solid mobile agent systems began to appear (Johansen et al. 1995; Johansen et al. 2002; Gray et al. 1998; Lange and Oshima 1998).

The mobile agent model may lead to more complex systems, but it brings several advantages (Lange and Oshima 1999) over the traditional models of distributed computing (like the client/server and distributed objects paradigms). Some are commented below.

- *Reducing network load and overcoming network latency*: distributed systems use communication protocols that often involve a considerable amount of message exchange through the network. When using mobile agents, however, the program migrates to the target host and hence the message exchange happens locally. The traffic is reduced to the agent migration and the network latency no longer applies.
- *Autonomous and asynchronous execution*: after migrating to another host, a mobile agent may become independent of the application that created it, enabling it to execute autonomously and asynchronously. Applications that rely on fragile connections may benefit from this because it is not necessary to maintain an open connection between the parts.
- *Dynamic adaptation*: mobile agents can receive information about their executing environment and react to changes.

In spite of that, there is no killer application for mobile agents, every mobile agent application can be implemented using traditional models. Even so, perhaps the most important contribution of the mobile agents is the introduction of a new paradigm that allows the construction of innovative systems and offers different approaches to known problems.

3 Mobile Musical Agents

We define a *mobile musical agent* (or simply *agent* from now on) as a mobile agent which participates in a musical process. It may do so by performing one or more of the following activities.

Encapsulating an algorithm: as a computer program, an agent can carry algorithms, in particular composition algorithms (Miranda 2001; Roads 1996; Rowe 1993). These algorithms may also require input data that may be carried with the agent or generated by it, allowing the agent to produce music autonomously.

Interacting and exchanging information with other agents: similarly to a situation where real musicians play together on a stage, several agents can interact with each other exchanging musical information.

Interacting with real musicians: an agent can receive commands or audio/musical data from a real musician. The commands could be as simple as notes played by the musician on a MIDI keyboard or new parameters for an algorithm executed by an agent. The agent could also receive the audio from an acoustic instrument and process or reproduce this sound.

Reacting to sensors: agents can receive commands from other non-agent programs. These commands could be triggered by sensors so that the agents could react to events in the real, physical world like the movements of a ballerina on the stage or the activity of the public in a museum.

Migrating: a migration process can be set off by the above actions. In other words, the agent can decide to migrate

- stochastically or deterministically, based on an algorithm;
- based on the interaction with other agents;
- based on the interaction with musicians;
- by reacting to sensors.

An agent that migrates resumes its performance when it arrives at its destination, which can be either in the same room or in another room, city, or country.

We can build musical systems using the model described. See a few examples below.

1. *Stochastic melodies:* in this system, the agents encapsulate a stochastic algorithm that generates a melody. A system component or one of the agents may work as a metronome, giving the other agents the correct timing. The result of the performance of several such agents in the same host would sound as synchronized stochastic music.
2. *Distributed performance:* each human musician is represented by one or more agents. Each agent receives the music played by the musician it represents and reproduces it in real time in the computer where it is hosted. In this system, a musician can be virtually present on more than one stage at the same time. These agents could be used to conduct a distributed performance: each musician in a different location receives agents representing other musicians in his computer and sends his agents to the computers of the other musicians.

3. *Collaborative music:* in systems such as DASE (`http://www.soundbyte.org`), users interact and exchange audio files through the network in order to compose a collaborative musical piece. A mobile musical agent system could use this same idea, except that the users would implement and dispatch their own autonomous musical agents, which would interact with each other.

4. *Interactive music system:* the association of mobile musical agents with interactive music systems (Rowe 1993) is natural, considering that an agent may receive musical information and respond to it.

5. *Distributed music:* consider a museum or exhibit hall equipped with several computers connected by a wireless network. Each computer could be equipped with motion sensors and host a few agents. The agents would communicate with each other and play a distributed music piece in a synchronized manner. A specific agent could receive information from motion sensors in order to follow a person who walks around the room (using its ability to migrate). The listener perception would be that part of the music is following him. Another part of the music, on the other hand, could run away from the listener, migrating to computers far from where the listener is. Sounds generated by the public (for example, a short speech) could be dynamically incorporated into the musical environment created by this system.

We have contacted composers and researchers of contemporary music that have already shown interest in studying the possibilities of the mobile agents application in music. This collaboration will be essential for us to discover the real potential of this technology.

The above examples also bring us computational problems related to real-time, latency, and quality of service. They represent important open research problems that are beyond the scope of our present work.

4 Andante

This infrastructure offers software components for a user to build applications similar to the examples shown.

This section describes the infrastructure by discussing the technologies employed, the architecture, and implementation details.

4.1 Technologies

The whole system is written in Java for the following reasons.

- *Platform independence*: we expect programmers, composers, and instrumentalists to use our system. For this to happen, we believe the system must run in distinct software and hardware environments such as the ones based on Linux (mostly used by programmers), Mac OS (mostly used by composers), and Windows (mostly used by instrumentalists). Java currently seems to be the best alternative to build systems that easily run in all these platforms.

- *Java Swing*: Java offers a solid library for the construction of graphical user interfaces. It is important for us to build platform independent interfaces quickly.

- *Multimedia support*: the official implementation of the *Java Sound API* (`http://java.sun.com/products/java-media/sound`) does not fully suit our needs yet, however, it allowed us to build the infrastructure prototype in a short time.

The sound generation is currently based on the MIDI classes provided by the Java Sound API, therefore it is based on the MIDI protocol. Nevertheless, we tried to avoid too much influence from this protocol because of its known limitations for sophisticated musical applications.

Although we have so far only used Java, we would also like to allow parts of the system to interact with components written in other programming languages. The reason for this is to make possible to use other technologies for sound generation other than the ones provided by the Java Sound API. For example, we have started experiments with the MAX/MSP environment. We are writing a Java class that communicates with a MAX/MSP patch. For this we are using the OpenSound Control protocol (Wright and Freed 1997) support for MAX/MSP built by the CNMAT (`http://www.cnmat.berkeley.edu/OpenSoundControl`).

We are also using the CORBA (OMG 2002) middleware, which allows programs written in different languages and running on different operating systems to communicate to each other seamlessly. All the communication among the components of our system is performed via CORBA. This will allow us to integrate the Andante infrastructure with systems such as CSound (Boulanger 2000), written in C, and Siren (Pope and Ramakrishnan 2003), written in Smalltalk.

Our infrastructure is built on top of the *Aglets Software Development Kit* (ASDK). Aglets (Lange and Oshima 1998) is a mobile agent system, written in Java and originally developed by IBM (`http://www.trl.ibm.com/aglets`). It is currently an open source project (`http://aglets.sourceforge.net`), offering libraries and applications to implement and manage Java mobile agents.

4.2 Architecture

An agent performs its actions in a heterogeneous computer network environment. The computers in this network must run a host software that we call *Stage*. This software is a component of the architecture, it represents a place where multiple agents meet and interact.

The Stage also offers the means for the agents to perform their actions. In particular, to produce sound, an agent needs to use the Stage's sound generation service. To provide this service, the Stage uses another component of the architecture: the *Audio Device*.

We have thus defined three key elements of the Andante architecture: the Agent, the Stage, and the Audio Device. Figure 1 depicts an abstract overview of the architecture.

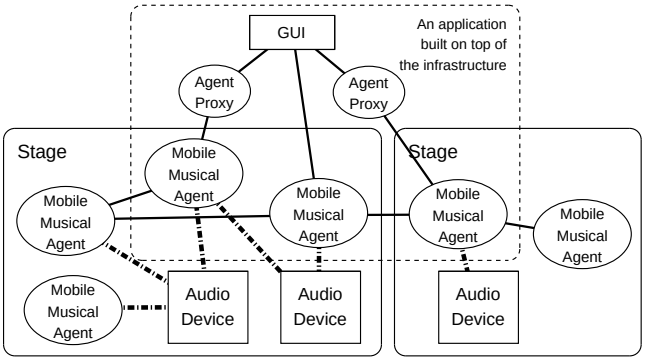

Figure 1: Architecture overview

An additional element, the GUI, is shown in the figure. It is not necessarily a component of the architecture, but it is the main component of applications built on top of the infrastructure and plays the important role of supporting human interaction with agents.

The application takes advantage of a fourth component of the architecture: the *Agent Proxy*. This element provides location transparency for the agents. When an agent migrates, it informs its new location to its proxy, which in turn is responsible for the communication between the agent and the GUI. The GUI may also choose to communicate directly with the agent, or to be the proxy for one or more agents itself.

4.3 Implementation

Figure 2 shows a UML (Booch et al. 1998) class diagram of the architecture. The `MusicalAgent` class represents the mobile musical agent and the `Stage` class is responsible for hosting agents in a computer. All the instances of these classes register themselves with the CORBA Naming Service (a centralized service) so that they can easily find each other in the distributed system. Both classes are built using the Aglets Software Development Kit, so they must be written in Java.

The Stage offers services for the agents, the most important are described below.

- `channel`: provides a channel of communication through which sound generation request messages are sent, similarly to the MIDI protocol (but not limited to it). Each Stage has several available channels, each allowing different settings (for example, the timbre used to play notes).

- `metronome`: provides an object which works as a metronome. This object receives registration requests from agents and sends the `pulse` message to the registered agents at a regular time. The time interval between pulses is determined by the metronome time signature and tempo properties, and all the registered agents receive the pulse at (almost) the same time.

To implement a new kind of agent, the user of the infrastructure must implement messages of the `MusicalAgent` class (some are already implemented). The most import messages are the following (see Figure 2 for their parameters).

- `play`: tells the agent to start or resume its performance.

- `stop`: tells the agent to interrupt its performance.

- `set`: sets the property `propName` to the value `propVal`.

- `pulse`: if the agent is registered with a metronome, it will receive this message at regular times, representing the pulses of a certain tempo and time signature. The agent is supposed to take an action when this happens.

- `dispatch`: tells the agent to migrate to the destination determined by `address`.

The following code shows the implementation of a sample Andante agent.

```
1  // Every mobile musical agent must extend
2  // 'MobileMusicalAgent'.
3  public class RandomMelodyAgent
4    extends MobileMusicalAgent
5  {
6    boolean _play = false;
7    short [] _cMaj = {60,62,64,65,67,69,71};
8    java.util.Random _rand;
9    Channel _channel;
10
11   // This message is sent right after the creation of this
12   // agent.
```

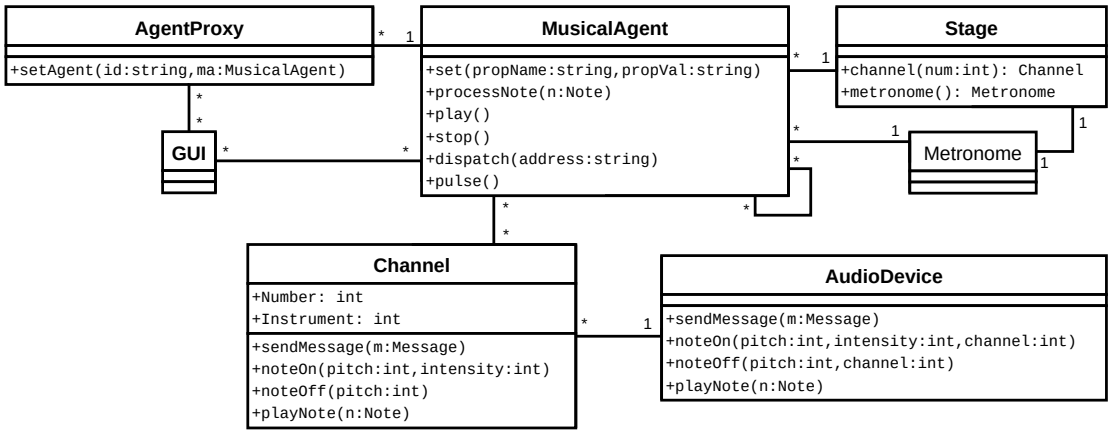

Figure 2: Architecture class diagram

```
13   public void init() {
14     _rand = new java.util.Random();
15     _channel = _stage.channel(1);
16     play();
17   }
18
19   public void play() {
20     _play = true;
21     int pitch, intensity, duration;
22
23     while (_play) {
24       pitch = _cMaj[_rand.nextInt(_cMaj.length)];
25       intensity = _rand.nextInt(128);
26       duration = _rand.nextInt(1000);
27       _channel.noteOn(pitch, intensity);
28       try {
29         Thread.sleep(duration);
30       }
31       catch (InterruptedException ie) {}
32       _channel.noteOff(pitch);
33     }
34   }
35
36   public void stop() {
37     _play = false;
38   }
39 }
```

This agent plays a random melody generated in real time. The following describes a scenario where the `RandomMelodyAgent` is used.

- *Agent implementation*: we already have an implementation, the `RandomMelodyAgent`. We need now an instance of this kind of agent, let us call it *rmAgent*.This instance represents a single agent.
- *Agent dispatch*: after its creation, *rmAgent* must be sent to an instance of `Stage`.
- *Arrival procedure*: when *rmAgent* arrives at the stage, a number of actions are carried out by the infrastructure. First of all, the inherited field `_stage` (referred to in the line 15 of the code) immediately begins to represent the stage. Then, the message `init` is sent to *rmAgent*. In this case, *rmAgent* gets a channel from the stage, which is now represented by the field `_channel` (also in the line 15), and sends the message `play` to itself.
- *Performance*: as a result of the `play` message, *rmAgent* begins its performance. It uses the operations of `_channel` to play random notes of the C major scale.
- *Agent control*: we have not done it here, but it is possible to implement a graphical interface to send messages to *rmAgent*. In this case, it could be used to send the `stop` message to the agent, or to dispatch it to another stage.

5 Applications

The following two applications were built using the implementation of the Andante infrastructure described in Section 4. They intend to give a concrete demonstration of the architecture viability.

5.1 NoiseWeaver

The NoiseWeaver application generates and plays stochastic music in real-time. It uses only one kind of agent: the *NoiseAgent*, which generates a single melody in real-time. In the generated melody, simulations of selected types

of stochastic number generators determine the pitch, intensity, and duration of the notes. We call this generators *noise* because they simulate the frequencies that occur in the spectrum of $\frac{1}{f^{\beta}}$ noises. For example, a certain NoiseAgent could play a melody in which the pitch of the notes is determined by a sequence of numbers that simulates a pink noise. This same agent could have a brownian noise sequence to determine the duration of the notes, and a white noise sequence for the intensity. Restating, what we call *noise* is a number sequence generated by a stochastic algorithm which is then mapped to musical parameters of a melody. See the *fractal algorithm* description in (Roads 1996) for more information on this composition algorithm.

The agents also register themselves with the Stage metronome service so that every agent in the same Stage gets synchronized.

The NoiseWeaver provides a GUI to control NoiseAgents hosted in several Stages. Before using the GUI, one must use the infrastructure services to create Stages and create and dispatch NoiseAgents. Figure 3 shows the window that displays the running Stages.

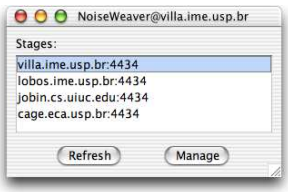

Figure 3: Available Stages

To control the NoiseAgents hosted in a Stage, the user has to select the Stage and press "*Manage*". A new window like the one in Figure 4 will open.

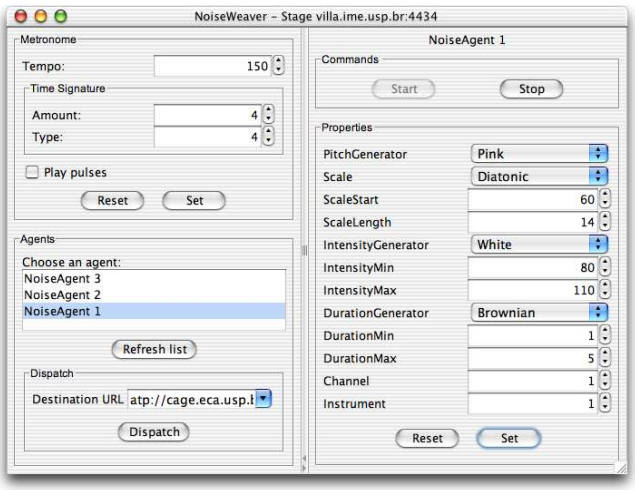

Figure 4: Stage control interface

This interface lets the user change the metronome and agents properties, even while the melodies are being generated.

The *Metronome* panel allows the user to define metronome parameters. *Tempo* is in beats per minute; *Amount* and *Type* define the time signature; and *Play pulses* determines whether the metronome plays a note at every beat. The *Agents* panel lists the NoiseAgents hosted in the Stage. The *Dispatch* refers to the selected agent in the list and it is used to tell the agent to migrate to another Stage.

When an agent is selected, the panel on the right hand side of the window is activated. This panel allows the user to change agent properties. The various properties that each NoiseAgent holds influence the generation of its melody.

On the *Commands* panel, we have:

- *Start*: tells the agent to start (or resume) playing.
- *Stop*: tells the agent to stop playing.

And on the *Properties* panel we have:

- Pitch: *PitchGenerator* defines the type of noise which generates the pitch of the notes. The selected noise is used to generate integer numbers which are then mapped to notes in the musical scale determined by the *Scale* property. *ScaleStart* is the point where the selected scale starts. The value 60 is equivalent to the middle C (as in the MIDI protocol). The *ScaleLength* determines the length (number of notes) of the chosen scale.
- Intensity: *IntensityGenerator* defines the type of noise used to generate note intensity (velocity, in MIDI terminology). The intensity is an integer between 0 and 127, *IntensityMin* and *IntensityMax* determine the interval of possible intensity values.
- Duration: *DurationGenerator*, *DurationMin* and *DurationMax* are the properties that define the note duration, they are similar to the intensity settings. The duration values are related to the metronome time signature.
- *Channel*: the channel to which the agent outputs its melody.
- *Instrument*: the instrument (patch) to be used to play the melody.

The possible values for the *Generator* properties are *Constant*, *White*, *Pink*, and *Brownian*. For *Scale*, we have *Diatonic*, *WholeTone*, *Chromatic*, *HarmonicMinor*, *HarmonicNatural*, *Pentatonic*, *Blues*, *PentaBlues*, and *Hirajoshi*.

Thus, using the NoiseWeaver, it is possible to have several NoiseAgents playing a noise-based melody each, and control the way all these melodies are generated using its GUI.

5.2 Maestro

The *Maestro* application is originally an extension of the *NoiseWeaver*. Instead of being controlled by the NoiseWeaver GUI, a distributed collection of NoiseAgents may be controlled by the Maestro, which in turn is controlled by a script. The main element of the script is the score, where a user can determine time-stamped changes in the agents properties. The Maestro offers a GUI to edit and run scripts shown in Figure 5.

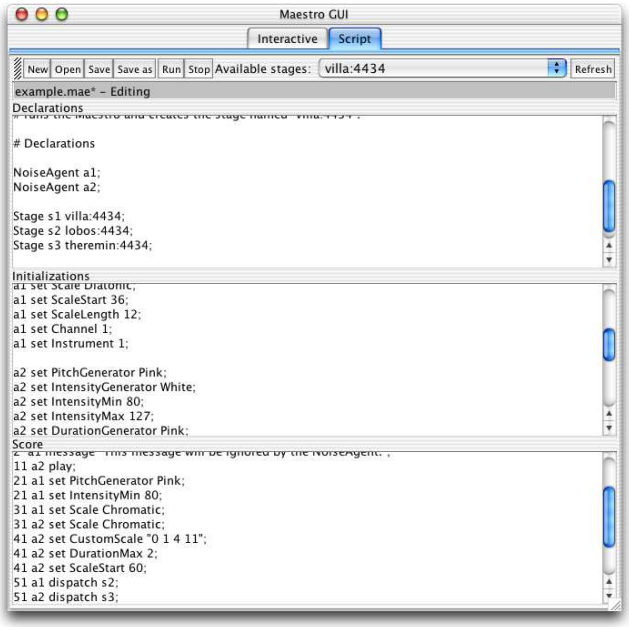

Figure 5: Maestro script interface

The script is composed of three sections, as shown in the figure. The *Declarations* section defines the stages and agents involved; the *Initializations* section lets the user set properties prior to the beginning of the score execution; the *Score* section is where the time-stamped property changes are defined.

It is important to state that the types of agents that can be controlled by the Maestro are not limited to the NoiseAgent. Any agent implemented in the Andante infrastructure may be used. Also, the so called "property changes" are actually messages of the `MusicalAgent` interface sent to a specific agent. So let us see a less informal description of the script.

<*Declarations*> is a list of at least one <*agent declaration*> and at least one <*stage declaration*>.
<*Initializations*> is a list of <*message*>.
<*Score*> is a list of <*time-stamped message*>.

<*agent declaration*>: '<*Java class*> <*id*>;'
<*stage declaration*>: '`Stage` <*id*> <*stage name*>;'
<*message*>: '<*id*> <*message name*> <*parameters*>;'
<*time-stamped message*>: '<*timestamp*> <*message*>;'

<*Java class*> is a name of an agent class.
<*id*> is a string not containing spaces nor ';'.
<*stage name*> is the name of a Stage.
<*message name*> is a message of the `MusicalAgent` interface.
<*timestamp*> is a positive integer.
<*parameters*> is a list of strings or quoted strings.

And, below, an example.

```
NoiseAgent a1;
NoiseAgent a2;
Stage s1 villa:4434;
Stage s2 lobos:4434;
-- # end of declarations
a1 set DurationMin 1;
a1 set DurationMax 120;
a1 set Scale Diatonic;
a1 set ScaleStart 36;
a1 set ScaleLength 12;
a1 set Channel 1;
a1 set Instrument 1;
a2 set PitchGenerator Pink;
a2 set Scale Diatonic;
a2 set ScaleStart 72;
a2 set ScaleLength 12;
```

```
a2 set Channel 2;
a2 set Instrument 36;
a1 dispatch s1;
a2 dispatch s2;
-- # end of initializations
1  a1 play;
5  a2 play;
13 a1 set IntensityMin 80;
13 a1 set Scale Chromatic;
17 a2 set DurationMax 2;
17 a2 set ScaleStart 60;
25 a1 dispatch s2;
25 a2 dispatch s1;
37 a1 stop;
37 a2 stop;
-- # end of score
```

In the Declarations section, two agents of the `NoiseAgent` type are created and two stages are defined. They are later be referred to by their identifiers (*a1*, *a2*, *s1*, and *s2*) in the other sections of the script. In the following section, some initial properties of the two agents are set (see Section 5.1) and the agents are sent to different stages. The Score section then defines the messages to be sent at specific moments. For example, at moment *1* the agent *a1* starts to play.

We have the collaboration of a composer who is currently writing script examples that will be published in our web site.

Interactive Interface. As another extension, we built a GUI very similar to the NoiseWeaver GUI, only that it can control any kind of agent. See Figure 6.

The interface has three important additional features.

- *Synchronize metronomes*: makes the metronomes of all stages send the `pulse` message at the same time.
- *Agent creation*: allows the user to create an agent with an associated identifier.
- *Generic property setting*: when the agent is created, the interface gets information about the agent's properties. It then uses this information to build the right hand side panel dynamically.

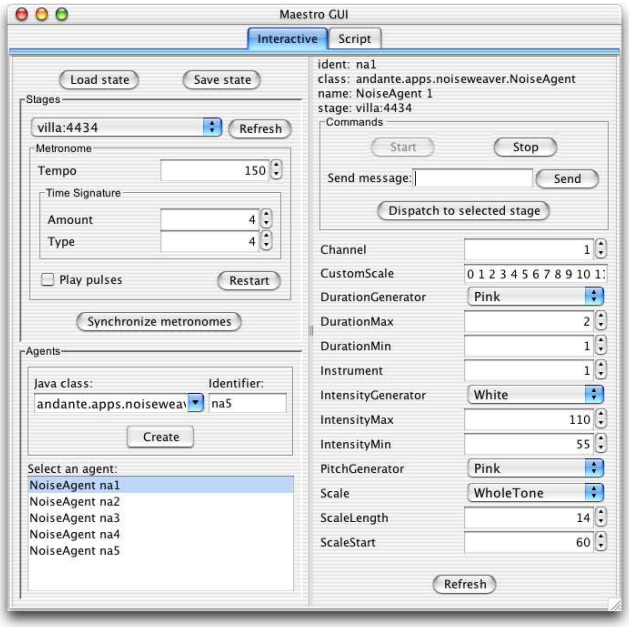

Figure 6: Maestro interactive interface

6 Conclusion and Future Work

The Andante project expects to be more than a computer system. We are hoping to create an open community where artists and scientists collaborate to create musical ideas, mobile musical agents, and to develop the enabling software infrastructure. To help this effort, we keep a site on the Internet: `http://gsd.ime.usp.br/andante`.

With the initial prototype of the infrastructure implemented, as shown in this paper, we are now moving the main focus of the project in the direction of musical creation. Our next steps will be guided by the interaction with composers.

We will continue the development of the infrastructure and the applications as we plan to add new functionalities and refinements.

Besides that, we plan to design and to build new applications to explore better the agent mobility and the human-agent and agent-agent interactions. The mobility may take place in a local network, where the computers are in the same location, and also in an environment where the computers are geographically distant. We intend to write musical pieces that explore these possibilities.

To make sure that the pieces will perform correctly, it will be necessary to implement the support for real-time and quality of service in the infrastructure. The idea is to allow the applications to define their system requirements (such as CPU load, memory, network bandwidth, etc.) so that they execute within acceptable time constraints. This support will be very important to attract the interest of more users.

Finally, it would be really interesting to build a programming environment based on visual models so that non-programmer users could implement their own musical agents. One such environment would be a challenging project related to Andante to be engaged in the future.

References

Booch, G., J. Rumbaugh, and I. Jacobson (1998). *The Unified Modeling Language User Guide*. Addison-Wesley.

Boulanger, R. (Ed.) (2000). *The CSound Book*. The MIT Press.

Gray, R. S., D. Kotz, G. Cybenko, and D. Rus (1998). D'Agents: Security in a Multiple-Language, Mobile-Agent System. In G. Vigna (Ed.), *Mobile Agents and Security*, Volume LNCS 1419, pp. 154–187. Springer-Verlag.

Johansen, D. et al. (1994, November). Operating system support for mobile agents. Technical Report TR94-1468, Department of Computer Science, Cornell University, USA.

Johansen, D. et al. (1995, June). An introduction to the TACOMA distributed system version 1.0. Technical Report 95-23, University of Troms, Norway.

Johansen, D. et al. (2002, May). A TACOMA retrospective. *Software - Practice and Experience 32*(6), 605–619.

Kotz, D. and R. S. Gray (1999, July). Mobile Agents and the Future of the Internet. *ACM Operating Systems Review 33*(3), 7–13.

Lange, D. B. and M. Oshima (1998, August). *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley.

Lange, D. B. and M. Oshima (1999, March). Seven good reasons for mobile agents. *Communications of the ACM 42*(3), 88–89.

Miranda, E. R. (2001). *Composing Music with Computers*. Oxford (UK): Focal Press.

OMG (2002, July). *CORBA v3.0 Specification*. Needham, MA: Object Management Group. OMG Document 02-06-33.

Pope, S. T. and C. Ramakrishnan (2003). Recent developments in Siren: Modeling, control, and interaction for large-scale distributed music software. In *Proceedings of the 2003 International Computer Music Conference*, Singapore.

Roads, C. (1996). *The Computer Music Tutorial*. The MIT Press.

Rowe, R. (1993). *Interactive Music Systems*. The MIT Press.

Ueda, L. K. and F. Kon (2003, August). Andante: A mobile musical agents infrastructure. In *Proceedings of the 9th Brazilian Symposium on Computer Music*, Campinas, Brazil.

Wright, M. and A. Freed (1997). Open SoundControl: A new protocol for communicating with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, Thessaloniki, Greece.