Ensemble: Implementing a Musical Multiagent System Framework

Leandro Ferrari Thomaz and **Marcelo Queiroz**
Computer Science Department – University of São Paulo – Brazil
{lfthomaz | mqz}@ime.usp.br

ABSTRACT

Multiagent systems can be used in a myriad of musical applications, including electro-acoustic composition, automatic musical accompaniment and the study of emergent musical societies. Previous works in this field were usually concerned with solving very specific musical problems and focused on symbolic processing, which limited their widespread use, specially when audio exchange and spatial information were needed. To address this shortcoming, *Ensemble*, a generic framework for building musical multiagent systems was implemented, based on a previously defined taxonomy and architecture. The present paper discusses some implementation details and framework features, including event exchange between agents, agent motion in a virtual world, realistic 3D sound propagation simulation, and interfacing with other systems, such as Pd and audio processing libraries. A musical application based on Steve Reich's Clapping Music was conceived and implemented using the framework as a case study to validate the aforementioned features. Finally, we discuss some performance results and corresponding implementation challenges, and the solutions we adopted to address these issues.

1. INTRODUCTION

In this paper we discuss implementation strategies and report recent experience with *Ensemble*, a musical multiagent framework first presented in [1]. The multiagent approach is well-suited for musical applications involving a number of autonomous musical entities that interact musically with one another, such as electronically-mediated collective performance [2, 3], automatic accompaniment and improvisation [4, 5], and biologically-inspired musical societies (used for studying emergent behaviors) [6, 7, 8, 9].

Although the literature on the use of agents in music is rather extensive, most of it deals with very particular problems [1]. Two previous works have much more general goals and are deeply connected to the present work, deserving special attention. The MAMA architecture [4] offers a framework for designing musical agents with interactive behavior based on the speech act theory, which communicate using MIDI messages to perform a musical piece. The SWARM Orchestra [2] is an user-extendable library that deals with large and complex populations (swarms), which may be used to control several musical and motion parameters simultaneously.

The *Ensemble* musical multiagent framework, which was first proposed in [1], builds up on the ideas of these two systems to define a generic, extendable, and configurable framework. Two general types of agents are considered in this framework: musical agents, which inhabit a virtual environment and interact with one another via sensors and actuators, and an environment agent, which controls the virtual environment and all interactions therein.

Musical agents are autonomous pieces of software that may embody interactive musical algorithms, or may also serve as virtual proxies to external agents, such as instrumentalists or even other musical software systems. They can also serve as sound outlets, capturing sound at specific positions in the virtual environment and sending them out for playback on a real listening space, such as a concert hall or an installation space, using either loudspeakers or headphones.

Interactions are modelled as events, which can be of several types, such as sound events, motion events, visual events and textual/symbolic messages, and each event type is controlled by an event server (which is part of the environment agent).

Musical agents can be specified using initialization files or can be created and modified at runtime. Agent design allows agent components, including sensors and actuators, agent reasonings and sound-processing engines, also to be added and removed at runtime, making this a pluggable framework.

Interfacing the architecture with popular sound processing languages and environments, such as Pd or Csound, is also a major concern. Currently, agent creation and modification, as well as on-the-fly control of agent motion, sensing and acting, can all be done using Open Sound Control (OSC) messages [10].

Ensemble extends the functionalities of [4, 2] by aggregating many novel features, such as multimodal communication (audio, MIDI and text-based) between musical agents, pluggable components for defining agents and physical characteristics of the virtual environment, and 3D sound propagation simulation within the virtual world. In particular, audio exchange between agents and a realistic treatment of space and acoustics, both poorly explored in previous works, are defining characteristics of this work.

This paper is structured as follows. Section 2 discusses the specific details of the implementation of the framework, and also the specification of agents and components by the user. Section 3 presents a concrete musical application of the system, based on Steve Reich's *Clapping Music*, as a case-study to illustrate the framework from the user point-of-view. Finally, some concluding remarks and pointers to further work are given in section 4.

2. FRAMEWORK ARCHITECTURE AND IMPLEMENTATION

This implementation was coded in the Java SE 6 language. Although Java performance limitations and poor sound processing support are well known to the community, this choice was made so that any musical applications programmed by the user could be run on distinct platforms. The JADE 4.0 multiagent middleware [1] was chosen for being a well-documented and well-supported multiagent platform.

This framework can be classified as a *white-box* framework [11], since the user is required to have some knowledge of its internal implementation. User specific implementations and extensions to the system need to follow a few internal conventions, since the framework acts as a main program, calling user-defined methods. Nevertheless, we provide a reasonable amount of reusable components (such as analysis and synthesis engines) which may ease considerably the specification of a musical agent by the user.

Simplified UML class diagrams for the *MusicalAgent* and the *EnvironmentAgent* can be seen in figures 1 and 2, respectively. These two kinds of agents are based on the *EnsembleAgent* class, itself a subclass of JADE's *Agent* class, which provides basic functionalities such as mechanisms for message passing and scheduling/executing concurrent activities, as well as the definition and control of agent life cycles.

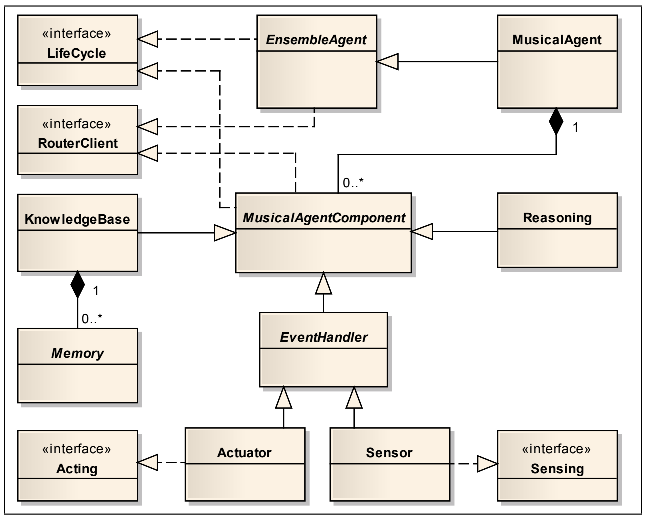


Figure 1. Class Diagram of the Musical Agent.

A *MusicalAgent* is composed of one *KnowledgeBase* object (for holding multiple data, such as I/O sound information) and possibly several *MusicalAgentComponent* objects, as shown in figure 1. These components can be of two types: *Reasoning* components, which are responsible for an agent's decision processes, and *EventHandlers*, i.e. *Actuators* and *Sensors*, capable of interacting with the environment through corresponding *Acting* and *Sensing* interfaces.

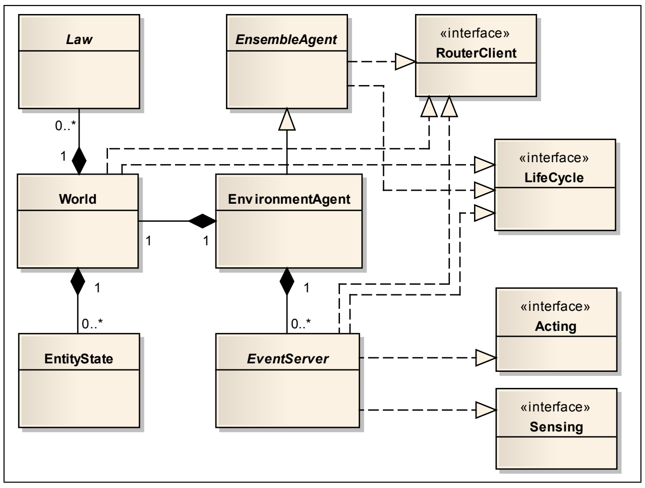


Figure 2. Class Diagram of the Environment Agent.

A special singleton agent, the *EnvironmentAgent*, represents the virtual environment and manages all interactions between *MusicalAgents*. As shown in figure 2, it is composed of a *World* object, for describing the virtual environment, and *EventServers*, for mediating event exchanges between *EventHandlers*. The *World* object contains the physical description of the space (number of dimensions, connectedness, boundaries) and also stores the current state of all entities (agent positions, motion intentions, sound produced and received, etc.) in *EntityState* objects. *Law* objects define the way the world state changes, i.e. they describe how to update the description in the *World* object given the last state, the current time instant and all actions currently performed by the agents. For example, realistic 3D sound propagation is defined by a particular *Law* object that receives all sound produced anywhere in the environment and delivers a specific mixture to each sound *Sensor* according to its position relative to each sound *Actuator*.

Major components of the framework (agents and their aggregated components) implement a *LifeCycle* interface. *LifeCycle* methods are: *configure()*, to set up configuration parameters before startup; *start()*, used by the framework to start each component; *init()*, user-specific initialization, implemented by the user and called automatically by *start()*; *stop()*, used by the framework to stop each component; and *finit()*, user-specific finalization, implemented by the user and called automatically by *stop()*. This approach provides greater flexibility when extending components, while ensuring the necessary control to the framework.

[1] Available at `http://jade.tilab.com/`. This and every other link mentioned in this text has been verified march 25th 2011.

2.1 Event Exchange

Ensemble supports two kinds of event exchange methods: *sporadic*, where events can be sent at any instant and rate (e.g. changing position of an agent or sending a MIDI or text message); and *periodic*, controlled by a synchronous communication process with a fixed exchange frequency, where in each cycle *Actuators* are requested to produce *Events* and *Sensors* receive corresponding *Events* (e.g. audio communication). Both types of event exchange methods are controlled by corresponding *EventServers*, and *Actuators* and *Sensors* interested in that event type (audio, for instance) are required to register prior to participating in the communication process.

The periodic exchange mode is controlled by state machines built into the *EventServer* and registered *Actuators*. State changes are regulated by the Virtual Clock service, responsible for timing the current simulation and scheduling tasks. Figure 3 shows an UML sequence diagram of a complete cycle of a periodic event exchange between an *EventServer* and an *Agent* equipped with a *Reasoning*, an *Actuator* and a *Sensor*.

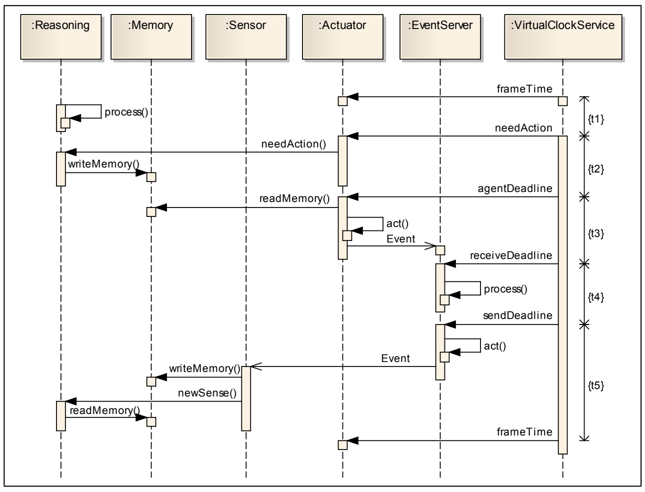


Figure 3. Sequence diagram of a periodic event exchange cycle. Time runs downward according to the timeline at the right of the figure.

At the beginning of each frame, there is a time interval for the agent to process any information that it wants to send in the next frame. When the deadline *needAction* is reached, the *Actuator* calls a *needAction()* method of the *Reasoning* responsible for writing the required information in the *Actuator*'s memory. The *agentDeadline* indicates that an *Event* must be sent through an *act()* method; failure to meet this deadline will result in no *Event* being sent (an empty audio frame, for instance). The *EventServer* waits for all arriving *Events* until the *receiveDeadline*, when it starts its own *process()* method, updating the *World* state and sending back response *Events* for registered *Sensors*, no later than the *sendDeadline*. When a *Sensor* receives an *Event*, it writes the corresponding information in its *Memory* and informs the *Reasoning* about it through the *newSense()* method. Finally, at *frameTime*, a new exchange cycle begins.

A *Reasoning's process()* method can be defined as an endless loop, or it can be triggered by the *needAction()* method. All deadlines are user-defined, as they depend on the amount of time needed for each *process()* method and for the communication between *Agents* and *EventServers*.

2.2 Agent Memory

An agent's *Memory* is the part of its *KnowledgeBase* used to store incoming and outgoing *Events*, analogously to a computer's memory. Every *Sensor* and *Actuator* has an associated memory, which is automatically filled in when a corresponding *Event* is received by a *Sensor*, and read from when an *Actuator* is asked to act. When a *Reasoning* component wants to send out some *Event* (sound, for instance), it stores the information in the corresponding *Actuator's Memory* and triggers the corresponding action. When an *Event* is received by a *Sensor*, it stores the information in its memory and lets all registered *Reasonings* know about it.

Memories are time-based, since all events have timestamps and durations, which are used to fill in the *Memory* or read from it. Although *Sensors* and *Actuators* follow a linear-forward time policy when accessing memories, there are many other components which may be interested in nonlinear or even random access, granted by generic *read(instant, duration, unit)* and *write(instant, duration, unit) Memory* access methods.

Two kinds of memories based on the same *Memory* interface were implemented. The simplest one is the *EventMemory*, which stores events in a double linked list, ordered by timestamp. Since search time tends to grow linearly with list size for regular linked lists, a simple heuristic was used to improve memory access performance for common situations. When processing audio or movement, sequential memory accesses are most likely to occur, and so the last seeked timestamp is used as a starting point for successive memory accesses.

A specialized *AudioMemory* was designed for dealing with audio data, which is implemented as a circular buffer of samples with a parameterized sample rate. This memory can be accessed using timestamps and also sample indices, using interpolation for continuous (floating-point) memory access. Linear interpolation is used by default, but non-interpolated or N-point interpolated accesses are easily configurable.

2.3 Agent Motion

A *MovementEventServer* is responsible for managing agent movement requests and updating agent positions. Agents equipped with movement actuators are able to change their position in the virtual environment using simple instructions such as WALK, ROTATE and STOP. Depending on the *World* definition (and its *Laws*) these may be used to instantly update an agent's position, to instantly start moving with a given velocity towards a certain direction, or more realistically, to define an acceleration (as if induced by a physical force) and let the *EnvironmentAgent* compute the corresponding trajectory using basic (i.e. Newtonian) mechanics. This is defined by a specific *MovementLaw*.

The rate at which the *MovementEventServer* updates its data about agent positions (within the corresponding *EntityState*) is defined by the user. At each update cycle, the server checks if any agent has pending movement instructions to be carried out. Friction can also be considered in the simulation, with user-defined friction coefficients. The *MovementEventServer* can check for obstacles, such as other agents or walls, thus restricting an agent's movement. Agents can be informed of the result of their movement requests (and their updated position in the environment) via specific *MovementSensors*.

A *MovementReasoning* was conceived to help design agent trajectories defined by waypoints and time-of-arrival constraints. This *Reasoning* sends out acceleration instructions to the *EventServer* through a movement actuator, and monitors the agent's actual position using a movement sensor.

2.4 Sound Propagation

Realistic and reliable 3D sound propagation simulation within the virtual environment was one of the central issues in the design of *Ensemble*. This corresponds to having each agent hear/sense exactly what it would in a real scenario, according to the positions of its sound *Sensors* and the positions of every sound *Actuator*, their corresponding velocities (with an implied Doppler effect), attenuation effect due to distance, sound shadows cast by obstacles, etc. This is of paramount importance when a sound design is going to be reenacted in a real listening environment, such as a concert hall or an installation, and the impression of a realistic spatial soundscape is intended.

There are a few technical issues involved in a realistic sound propagation simulation that will be discussed in the sequel. First of all, it should be noted that sound is only perceived at sound *Sensors* (and not anywhere in the space), and so a simulation of wave equations on a discretized grid representing the space would be computationally prohibitive and also useless for the most part. Instead, sound propagation is considered independently for each pair (sound *Actuator*, sound *Sensor*). Global simulation parameters such as speed of sound, attenuation due to distance and frequency filtering due to the environment can all be configured by the user.

The *SoundEventServer* is a periodic process that is required to deliver to each sound *Sensor* one audio frame per cycle, representing all incoming sound at the current position of the *Sensor*, which is allowed to vary continuously. This means that, for each sound sample of this frame corresponding to timestamp t, the *Sensor* S has a different position $Sp(t)$, and the same is true for every other sound-producing *Actuator* in the environment. So, for each timestamp t of this audio frame (to be delivered to a particular sound *Sensor*), the event server has to go through all sound *Actuators* in the environment and find out, for each sound *Actuator* A with an independent trajectory $Ap(\cdot)$, when did it produce sound that arrived at position $Sp(t)$ at time t.

Considering that sound travels in a straight path with constant speed c, the problem is to find an instant $d(t)$ in the past such that the path from position $Ap(d(t))$ to $Sp(t)$ takes exactly $t-d(t)$ time units, or in other words, to solve the following equation in the variable d for each given t:

$$Sp(t) - Ap(d) = c(t-d).$$

Since the functions $Sp(t)$ and $Ap(t)$ have simple analytic derivatives (according to the Newtonian equations), the Newton-Raphson method provides a quick way to find the solution $d(t)$ for each S, t and A. This solution for a timestamp t can be used as a starting point when finding the solution for the next timestamp $t+\Delta$, whose solution $d(t+\Delta)$ is likely to be close to $d(t)$. Experimental tests showed that, with this initialization, it takes about four iterations for the Newton-Raphson method to find $d(t)$ within a precision of 10^{-9} seconds.

Despite Newton-Raphson's efficiency, it should be noted that this problem has to be solved once for each sample n of each sound *Sensor* S and for each sound *Actuator* A, with a total of $(\#\textit{Sensors}) * (\#\textit{Actuators}) * (\textit{FrameSize})$ calls to this function. For instance, in a very simple setting of two *Sensors* and two *Actuators* using a *FrameSize* of 100 ms with a 44.1 kHz sample rate, it would take 17640 function calls or about 70560 Newton-Raphson iterations to complete each processing cycle, which gives less than 1.5 μs of CPU time per Newton-Raphson iteration, only for sound propagation. As the number of *Sensors* and *Actuators* increase, the chance of the sound event server losing its periodic deadline becomes a threat.

To minimize this problem, a polynomial interpolation method combined with the Newton-Raphson method was used. This approach finds the precise values of $d(t)$ for the first and last sample of each sound frame, and for as many points in between as necessary according to the polynomial degree chosen. Then interpolation using Neville's algorithm is used to obtain $d(t)$ for the remaining samples. Experimental tests with a frame size of 100 ms showed that quadratic interpolation (3 points per frame) provide values of $d(t)$ within less than 10^{-5} seconds of their correct values, corresponding to subsample accuracy, even when *Sensors* and *Actuators* change their accelerations within the considered audio frame. Cubic interpolation (4 points per frame) drives errors down to 10^{-8} seconds or 0.000441 in terms of sample index.

Figure 4 shows performance measurements [2] made with the framework; these values correspond to the time dedicated to computing the sound propagation between one *Actuator* and N *Sensors*, expressed as a fraction of the frame size. As expected, values grows roughly linearly as a function of $(\#\textit{Sensors}) * (\#\textit{Actuators})$ for each fixed framesize, until a limit of operability is reached and the computation breaks down, meaning that not every sound produced gets propagated to every *Sensor*. This limit of operability (indicated in the figure by small boxes) increases with framesize, and for the particular equipment used in this experiment, frame sizes between 100 ms and 250 ms seem to offer a reasonable tradeoff between latency and stability for $(\#\textit{Sensors}) * (\#\textit{Actuators}) \leq 40$.

It should be noted that this approach do corresponds to a realistic sound propagation simulation that includes the

[2] Test were conducted using a MacBook Pro with a 2.7 GHz Inter Core 2 Duo processor and 4 GB of memory, running Mac OS X 10.6.

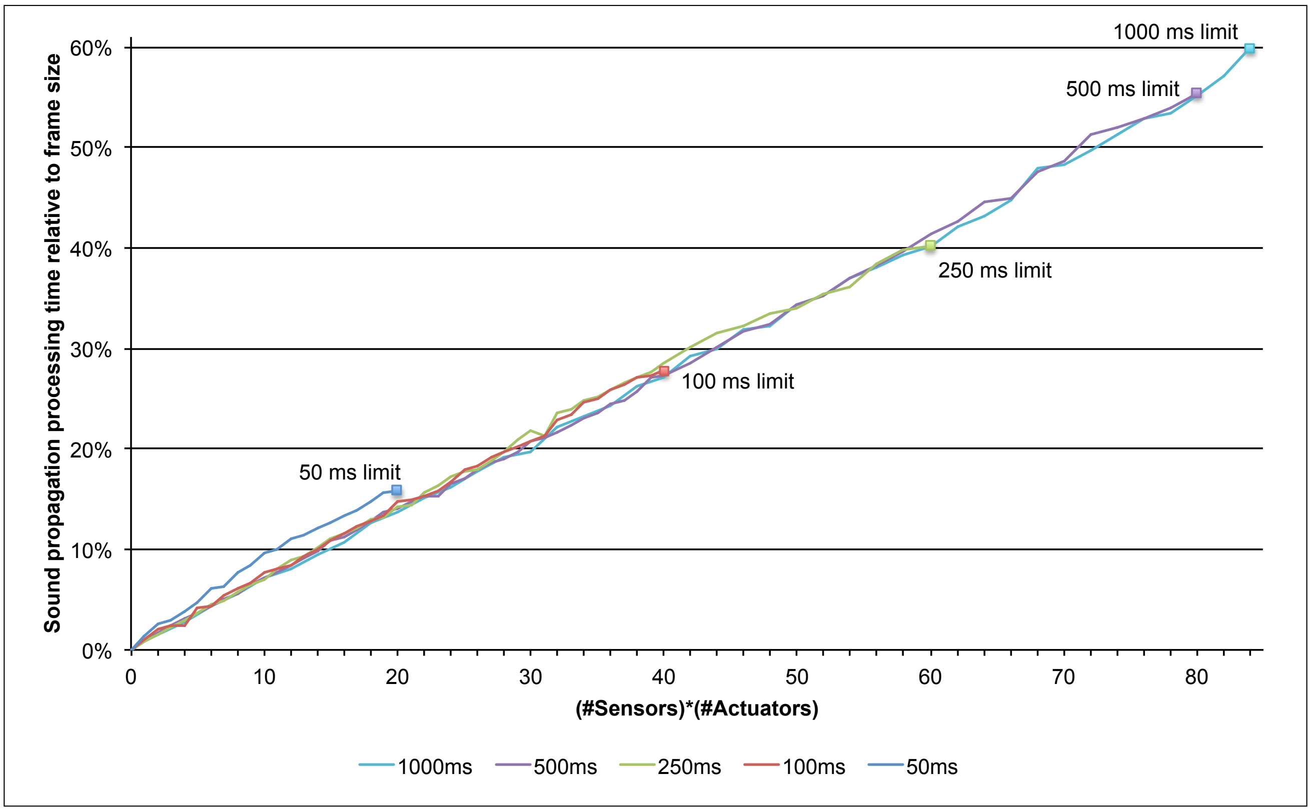


Figure 4. Sound propagation processing time for different frame sizes.

Doppler effect, since values of $d(t)$ depend on continuously changing values of position and speed of the *Sensors* and *Actuators*. Therefore, approaching targets imply compressing wavefronts, and distancing targets imply expanding wavefronts. Frequency shifts are a direct consequence of the definition of $d(t)$ above.

Since a single agent may have multiple sound *Sensors* positioned at different points of its virtual body, a sound wave has different arrival times for each *Sensor*. This allows the use of multi-sensor listener agents that capture 3D soundscapes and send them for external multi-channel playback. It also allows *Reasonings* to use these multiple inputs for source position identification, source separation, etc.

Last but not least, it should be noted that realistic sound propagation is but one of many alternatives in the design of acoustical laws for the virtual world. Some musical multiagent designs [6, 7, 8] rely on different formulations for the virtual space, such as discrete 2D spaces with planar or square wavefronts. Unearthly and bizarre sound propagation schemes could easily be implemented and used for sound experimentation, for instance sending out different frequency bands in different directions with different speeds spreading out from a single sound source.

2.5 Interfacing

Ensemble was designed to allow flexible implementation of musical multiagent applications, with the intention of generalizing from examples found in the literature. Nevertheless, interfacing the framework with other sound-processing and music-processing programs is beneficial for several reasons: it extends the available functionalities of the framework, it affords code reusability, and it improves user comfort, by allowing part of the application to be developed using a language or environment of the designer's choice.

We will discuss in the sequel several aspects of interfacing with *Ensemble* that we consider fundamental: interfacing with specialized libraries, interfacing with general sound-processing programs, user interfaces and audience interfaces.

Interfacing with external libraries

Two external libraries for audio processing were incorporated into the framework: *aubio*[3] and *LibXtract*[4]. Essential functionalities for audio processing such as FFT, digital filters and feature extracting functions can be used transparently when designing agents, reasonings, analysis and synthesis engines through these libraries.

Since these libraries are implemented in C and are platform-dependent, pre-compiled modules were created for the three most common operating systems used by musicians (MS-Windows, MacOS and Linux), which are accessed using Java Native Interface (JNI). An Abstract Factory Design Pattern approach ensures that other libraries can be incorporated in the framework when needed.

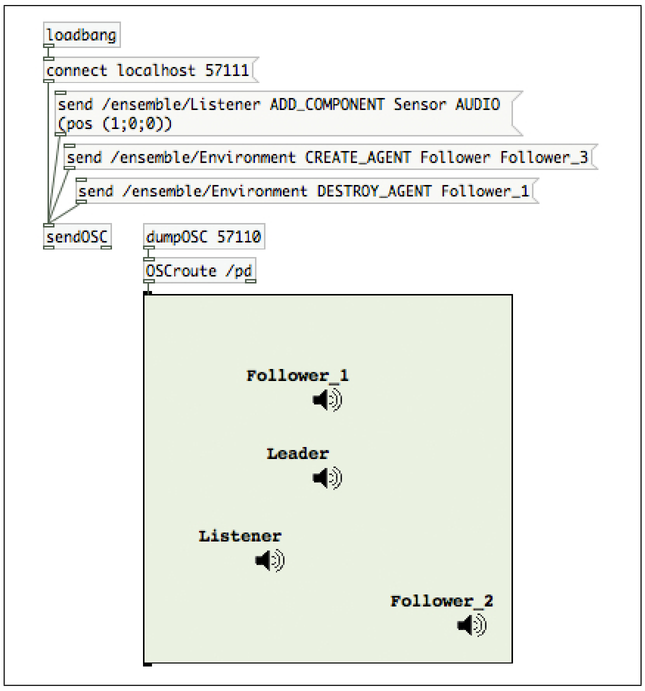
Interfacing with other programs

Every agent and component of the system is able to receive and send messages by means of a Router Service. This service, accessible through the *RouterClient* interface, is

[3] Available at `http://aubio.org/`.
[4] Available at `http://libxtract.sourceforge.net/`.

responsible for delivering every message to its correct recipient, using the internal JADE mechanisms for message exchange. The address scheme is based on a string containing the names of the system, agent and component. For example, one can send a message to a sound sensor belonging to a musical agent by directing the message to `/ensemble/pianist/right_ear`, or to an external program, for instance Pd, using the address `/pd`.

Open Sound Control (OSC) [10] is a message-exchange protocol specialized in musical applications, used and understood by major musical softwares. The routing mechanism of the multiagent framework can also be used to send and receive OSC messages through a special *RouterAgent* that implements an OSC server. For example, a Pd external was implemented that works as a graphical representation and user interface for an example 2D virtual world, using only OSC messages. Through it the user is able to see and interact with agents, send messages to start and stop sound processes, and place or move himself/herself within the environment to listen through headphones to a binaural version of the simulation.



```
loadbang
connect localhost 57111
send /ensemble/Listener ADD_COMPONENT Sensor AUDIO
(pos (1;0;0))
send /ensemble/Environment CREATE_AGENT Follower Follower_3
send /ensemble/Environment DESTROY_AGENT Follower_1
sendOSC
dumpOSC 57110
OSCroute /pd
```

Figure 5. Pd interaction with *Ensemble* using OSC.

Figure 5 shows an example of a Pd patch that interfaces with *Ensemble* by means of the OSC protocol. Two Pd objects[5] are needed for the communication to take place: *dumpOSC*, used for receiving OSC messages, and *sendOSC*, used for sending messages. These string messages must be parsed, using the *OSCroute* object, and interpreted by the Pd patch according to the user application. We developed a Pd external, *ensemble_gui*, which is a two-dimensional graphical representation of an agent's position on a rectangular virtual world. This external object processes commands sent by the *MovementEventServer* to update the graphical interface. The figure also illustrates some example messages that can be sent to *Ensemble* to create/destroy agents or to add audio sensors, all of which are possible at runtime.

[5] Bundled with Pd-extended, available at `http://puredata.info`.

Interfacing with a user or sound designer

Applications using *Ensemble* can be built using a single XML file, which contains parameters defining world components and agent components, as well as starting points for processes and rules for updating every aspect of the system. These XML constructs are meant to be simple indications of the methods, components and parameter values of the framework that will be used in actual simulation, and so they are much simpler than actual Java programming. Whenever an application relies on pre-built agent/world components, the user is able to use the XML file to assemble agents, plug-in their components (sensors, actuators, reasonings, etc), define world parameters and laws, trigger the start of the simulation and also to control the system at runtime using nothing but XML commands.

Extending the system is possible by either programming new components in Java, or alternatively by interfacing with other programs as already discussed. This second alternative is particularly interesting when designing graphical user interfaces for real-time user interaction, which is probably easier to do in Pd than for instance in Java.

Interfacing with real listening spaces

Nothing in this discussion would make sense if there were no channels to peep into or eavesdrop on the virtual environment while simulation is going on. Graphical user interfaces can be used to see the motion of agents as discussed above, while more advanced image processing techniques might also be considered for rendering spatial representations of the virtual world. But getting sound out of the virtual environment is the first and foremost goal when designing a musical multiagent application.

Two *Reasonings* were implemented to provide audio input/output using a regular audio interface hardware, so the user can hear what is happening inside the virtual environment and interact with it by inputting sound. The Jack audio system[6] was chosen as the audio library for this task for its low-latency, portability and flexibility, allowing a finer control of timing and also the use of multiple channels of the audio interface (both impossible with Sun's current Java Sound implementation). Jack requires the use of JNI, meaning that a library must be compiled for each operating system.

With Jack, it is possible to route audio channels between supported applications (*Ensemble* included) and also to/from an external audio interface. Thus, an external application such as Pd ou Ardour can export audio signals that are fed into *Ensemble* through a Musical Agent (using the *JackInputReasoning*), which may then be used as input to a musical Reasoning, or it may be propagated in the virtual environment through a sound *Actuator*. Any Musical Agent within *Ensemble* may likewise export audio signals using the *JackOutputReasoning*.

[6] Available at `http://www.jackaudio.org/`.

One consequence of the periodic event exchange approach is that a delay of two frames plus the delay of the audio interface itself is introduced when sound is captured and played back in the virtual environment; sound exported from the virtual environment is subject only to the delay of the audio interface.

3. CASE STUDY: CLAPPING MUSIC

In order to test some recent advanced functionalities of the framework, a relatively complex musical application was conceived. The starting point was the musical piece called *Clapping Music*, written in 1972 by Steve Reich. In this minimalist piece, a small rhythmic pattern is repeatedly clapped by two performers. While the first one goes on repeating the exact same pattern, the second one circularly shifts the beginning of the pattern one beat to the left, every 8 (or 12) repetitions, until they are once again synchronized, after 96 (or 144) repetitions. Figure 6 shows the pattern and its first shifted repetition.

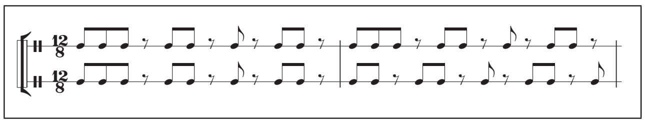


Figure 6. Clapping Music original pattern (first measure), and counterpoint produced with the first shifted pattern (second measure).

In the transposition of the piece to a musical multiagent application, some complicating hypotheses were introduced. First, the rhythmic pattern could be the one proposed by Reich, but it could also be randomly generated if the user so wished. Second, multiple agents (virtual performers) would be involved, with separate entries for each one, and also independent shifting patterns (described by user-controlled parameters). Third, and perhaps most importantly, agents would not be allowed to communicate by any means other than audio, i.e., we excluded the possibility of visual cues or trigger messages that would facilitate synchronization between agents. Everything would have to be done using audio analysis (i.e. finding out what the pattern is, when did it begin, etc.) and audio synthesis.

Three types of agents were implemented: a *Leader* agent who is responsible for proposing the pattern; a *Follower* agent who will try to discover the pattern and then start to play it, shifting each repetition by a certain amount; and a *Listener* agent who copies the output to an external user. At runtime, the *Leader* agent defines a rhythmic pattern (set by the user or randomly created based on the number of desired beats, bpm and wavetable), and start repeating it, accentuating each first beat of the pattern as a cue for other agents to pick up where it starts. Much more complicated pattern-discovering strategies could also be implemented without using such a cue, but they would inevitably prohibit patterns which contain repetitions of smaller sub-patterns or motives, such as AA or AAB patterns, for instance.

The *Follower* agent has two states: analysis and playing; in the analysis state, it must detect onsets as each audio frame arrives in its sensors, and create a list of timestamps and intensities. Real-time onset detection and other signal processing calculations (like FFT and RMS) used the *aubio* library. As soon as the second repetition starts, the agent has the pattern and is ready to play, but since it can only produce output for the next audio frame, it will wait for the third repetition of the pattern to enter with its shifted version.

Agent movements were defined to graphically illustrate the amount of shifted beats of each *Follower* with respect to the *Leader*. While the *Leader* stays at the center, the *Followers* perform an orbital-like motion around it based on their current shift value. Whenever a *Follower* starts a new repetition with a new shift value, it will walk towards a new orbital position, and all agents align when a cycle is completed (this may take much longer than the 144 repetitions of the original piece). Agent positions also influence how the user (through the *Listener* agent) will hear the piece, since sound propagation takes all agent positions into account, with corresponding delays, attenuation and filtering.

The flexibility in the application setup invites experimentation with different parameters values, including the amount and periodicity of shifting and the pattern itself, which brings about a myriad of intricate interweaving patterns. A curious effect due to spatialization is that followers far away from the leader will never play in perfect synchronism, since sound waves take some time to propagate, implying that the pattern is recognized with a certain delay.

An important fact about this application is that it uses mainly built-in general purpose *Ensemble* componentes. An XML configuration file is used to assemble all components and set up the simulation, and only one customized Reasoning had to be implemented in Java. Both files account for less than 500 lines of code, including comments.

4. CONCLUSION

This paper presented an updated account on the development of *Ensemble*, a framework for musical multiagent systems. Much has been improved since the first implementation presented in [1]: the current framework is more flexible, allowing integration with external libraries and programs, more user-friendly, allowing the specification of applications without Java programming and also the use of external graphical user interfaces, and also shows an increased level of realism and better performance in the sound propagation simulation, due to physical simulation and major redesign in the internal data structures, such as agent *Memories* and virtual world *Laws*.

The choice of Java, motivated by the fact that it is a general-purpose, platform-independent language with wide availability and support, has also had its drawbacks, some of them predicted and others not so much.

Predictably, Sun's Java Sound API implementation is not suitable for a more demanding audio application. For instance, depending on the operating system and audio interface driver implementation, one cannot address a specific audio channel of an external audio interface. This difficulty has been overcome through the use of PortAudio, which, although it requires pre-compiled modules to work on every platform, guarantees full access to most sound peripherals.

Java's garbage collection mechanism can sometimes interrupt important time-constrained operations of the framework, such as the periodic event exchange, or important audio processing operations, such as the sound propagation simulation. Since one cannot control when the garbage collector will be called, the framework is bound to loose some audio frames whenever the system becomes overloaded. Using Java Real Time might be a way to solve this problem, since processing start times and deadlines could be enforced by a real time operating system.

Also, the first few runs of each method were observed to be slower than subsequent runs, since the Java Interpreter always tries to execute code without compiling it, and only decides to natively compile some code excerpt after it detects intensive repetition. This problem was circumvented by creating a warm-up repetition routine for computer intensive methods.

The object-oriented approach used in the modeling and implementation of the architecture implied a great deal of object creation and destruction, on several levels of abstraction. These operations are somewhat expensive for the Java Virtual Machine, since memory need to be allocated and constructors/destructors called. Some time-constrained methods which deal with a lot of data, like the sound propagation simulation, are heavily hit by this fact. In order to increase performance, some coding techniques not much in line with the object-oriented paradigm were applied, like reusing the same object and minimizing the number of calls to a method.

Ensemble, in its current version, may be used to reproduce many kinds of musical multiagent applications, such as those discussed in [1]. Some performance improvements, relative to memory usage and synchronism between state machines, are already scheduled for implementation. These improvements are expected to allow the framework to work with lower latencies and an even larger number of agents.

The framework code, as well as example applications, is open-source and freely available on the web[7]. There is also a step-by-step tutorial on how to build a simple application with existing components. Documentation is expected to significantly improve in the near future.

Acknowledgments

This work has been funded by CAPES, CNPq and FAPESP (grant 2008/08632-8).

5. REFERENCES

[1] L. Thomaz and M. Queiroz, "A framework for musical multiagent systems," in *Proc. Int. Conf. Sound and Music Computing*, Porto, 2009, pp. 213–218.

[2] D. Bisig, M. Neukom, and J. Flury, "Interactive swarm orchestra-a generic programming environment for swarm based computer music," in *Proceedings of the International Computer Music Conference. Belfast, Ireland*, 2008.

[3] M. Spicer, "AALIVENET: an agent based distributed interactive composition environment," in *International Computer Music Conference*, 2004, pp. 1–6.

[4] D. Murray-Rust, A. Smaill, and M. Edwards, "MAMA: An architecture for interactive musical agents," in *Proceeding of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy*. IOS Press, 2006, pp. 36–40.

[5] G. L. Ramalho, P. Y. Rolland, and J. G. Ganascia, "An artificially intelligent jazz performer," *Journal of New Music Research*, vol. 28, no. 2, pp. 105–129, 1999.

[6] P. Dahlstedt and M. Nordahl, "Living melodies: Coevolution of sonic communication," *Leonardo*, vol. 34, no. 3, pp. 243–248, 2001.

[7] K. McAlpine, E. Miranda, and S. Hoggar, "Making music with algorithms: A case-study system," *Computer Music Journal*, vol. 23, no. 2, pp. 19–30, 1999.

[8] J. McCormack, "Eden: An evolutionary sonic ecosystem," *Advances in Artificial Life*, pp. 133–142, 2001.

[9] M. Gimenes, E. Miranda, and C. Johnson, "The development of musical styles in a society of software agents," in *Proceedings of the International Conference on Music Perception and Cognition*, 2006.

[10] M. Wright and A. Freed, "Open sound control: A new protocol for communicating with sound synthesizers," in *Proceedings of the 1997 International Computer Music Conference*, 1997, pp. 101–104.

[11] R. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[7] Available at `http://code.google.com/p/musicalagents/`.