A Distributed, Object-Oriented Framework for Sound Synthesis

Gerhard Behles
Electronic Studio at TU-Berlin
EN - 8, Einsteinufer 17-19
10587 Berlin, Germany
gb@kgw.tu-berlin.de

Peter Lundén
Royal Institute of Technology
7TH Box 700 14
S-100 44 Stockholm, Sweden
lunde@kacor.kth.se

Abstract

Object-orientation increases portability, flexibility and reuse of software components; distribution improves performance, scalability and collaboration. We propose a distributed object computing (DOC) framework for the development of sound processing software on workstation- and personal computers, which apart from providing low-level support for object distribution, also mediates between program parts that have been written in different languages. The system's design is based on a generalization and unification of concepts from the various prevalent paradigms of computer music.

Introduction

Computer music software projects distinguish themselves from one another by scope and size. On the one hand, there are those which provide small solutions to specific problems, solutions generally conceived and realized by individuals; on the other hand, there are large projects which require collaboration and often involve hard-ware design. Both approaches to software design share practical shortcomings.

Intelligent solutions provided in small computer music software projects often cannot be fully appreciated by users because of a lack in software infrastructure. With "infrastructure", we reference program components which are not really specific to the project but essential, for the full exploitation of its potential; compatibility for high-level interacting with other programs may be regarded as a prime example.

Large-system design projects generally employ some variety of modular realization, dictating categorically a specific paradigm of music production on both end-users and on software contributors. Consequently, valuable implementations are constrained by a paradigm and cannot be accessed by users who prefer to work within a different one.

We propose conceptually small projects which interface, ultimately resulting in a larger, scaleable and portable system. We are convinced that this concept can be achieved on standard computer and networking systems, employing distributed object computing technology (DOC).

An enumeration of the demands on a framework for computer music software and an introduction of DOC follows; additionally, an identification of the essential concepts in computer music and their respective generalizations in our design is given.

Demands on the system

Ideally, the design concept will unify projects which are conceived and developed independently (i.e. by different authors, at different times, for different purposes). To a large extent, design and implementation of signal processing techniques can be independent of the applications using them (e.g. filtering, dynamics modification, spatialization or phase-voicing techniques). As general purpose program components, they might be employed in real-time applications, as live-electronics, in MUSIC-n type composition languages or in digital audio systems. Equally, software which deals with high-level organization of musical procedures should be open to extension by low-level components.

"Universal applications" implies neutrality towards invocation schemes and distribution. End-users and programmers should be able to create and replicate components, into any memory space, at will. Component interaction should occur in a standardized fashion, regardless to how they are distributed among processes and machines.

Where efficient signal-processing code is required, it's common practice to use low-level programming languages and conversely, to use high-level, interpreted languages for musical organization tasks. Accordingly, the system must also mediate between programs written in different programming languages.

Distributed object computing

Related goals have been pursued, in recent years, in other domains of computer science. DOC technology has emerged to address these problems. DOC applied object-oriented concepts, specifically, encapsulation,
inherence and strict typing to distributed processing. This approach provides abstractions of the communication mechanisms used in distributed systems and softly allows high-level interaction between applications which are distributed arbitrarily among computers and processes. DOC systems support separation of program interfaces and implementations, and accordingly, increasing their extensibility and portability.

Many similar approaches are found in specific manufacturers (OLE / Microsoft) and thereby fail to comply with portability goals, as stated above. The JAVA language can be regarded as a DOC system, but it remains doubtful whether a single language can satisfy the demands of the gamut of computer music applications.

The CORBA industry standard defines properties of DOC systems which are independent of vendors, implementation languages and communication standards. CORBA products are available from a variety of vendors and Xerox PARC distributes a system called ILU (Inter Language Unification) with a subset of the CORBA functionality in the public domain.

To make use of a CORBA system, programmers must provide an interface file containing declarations of all data types used by their application. In particular, the interface declares all classes along with their methods and inheritance relations. This interface is written in a simple meta-language (Interface description language) which resembles C++ declaration syntax. From this specification, the DOC system can render language-specific header-files and code for "surrogate" and "stub" classes to all supported languages. The application programmer supplies implementations of "true" classes by deriving subclasses from the surrogates.

A surrogate object takes the place of a true object residing in different address space (or process or computer) by communicating requests from calls on its methods to the true object and returning results to the caller. A method call on a remote object is as simple as a call on an object in the same memory space and no performance loss is incurred in the latter case.

Essential concepts

The design of a DOC-system for computer music applications must take into account the diversity of applications, and environments encountered within the field. To be generously versatile, the system's design must follow a "least common denominator" approach. An investigation of these paradigms in computer music and general digital audio systems reveals recurring concepts which differ essentially in name and can be generalized for use in the system.

Processing units

In most systems, signal processing is carried out in conceptual and computational modules. This concept borrows from analog hardware circuitry, catering to physical inputs and outputs. This design concept is prevalent in the digital audio domain in three main variants:

- a) With respect to target-group needs, commercial harddisk-recording and editing system manufacturers design their products to resemble the traditional recording studio, where signal modification is accomplished in "effect boxes" and in "mixing console units" (e.g. Digidesign's TDM Bus / Pro Tools).
- b) MUSIC-n family computer music languages (e.g. Csound, Nyquist, CLM, etc.) and the MAX language have adopted the analog synthesizer module metaphor
- c) Other computer music environments (CARL, CEP, etc.) provide a suite of sound processing programs which can be applied in sequence.

These implementations share conceptual similarities, defying the reasoning which locks them away in a single application paradigm. Mapping signal processing units to object-oriented classes is straightforward. A generator class represents units designed to relay specific kinds of signals to others; while signal processors, with multiple and varied output "sockets", are represented as classes which inherit from a multitude of base-classes. Generator subclasses are predefined for a variety of signal-types: sampGen delivers time-domain samples, spectrumGen produces frequency-domain (data as obtained from short-time fourier-transform) and procGen are available when phase-voicer representation is desired. New generator types (e.g. wavelets-representations, compressed audio or video formats) can be defined by application programmers and integrated smoothly into the environment. To implement a new generator, application programmers derive pre-defined classes into more specific ones which contain the required functionality. The phone class, for example, is a subclass of sampGen which delivers samples from an audio input. Most MUSIC-n type languages evolve unit generators with instrument calls, but do not provide persistent units. Sound processing that should conceptually occur after the instrument (e.g. reverberation and spatialization) must be carried as part of the instrument call, a very inefficient strategy when linear processes are modelled. DSP hardware-based systems generally employ a more static scheme of allocation and tend to announce creation of new software units with a pause in the output stream or, even worse, with a click. To allow for complete flexibility in object instantiation, the system provides a factory class which can be used to create new objects at any time during the synthesis process and in any of the address spaces used by the application.

Patch/ routing

The paradigms mentioned have individualized models for signal flow. Systems described under a) handle signal flow in a "routing" fashion, whereby, processing units communicate signals via audio buses. In contrast, the analog synthesizer approach directly connects module sockets, without the bus "detour". In MUSIC-n languages, an analog synthesizer "patch" is represented as a directed acyclic graph (DAG) of unit generators.
While in category systems, signal flow is handled through shared files or UNIX pipes. Each file describes the execution sequence, and implicitly, the flow of signals through the various processes.

The most general approach is to handle signal flow in a directed graph of processing units; but routing and signal communication, through shared files and pipes, are easily mapped into a graph structure.

Objects which require signal input to perform function are modelled as consumers in the system. A consumer maintains any number of input objects, each of which represents a connection to a generator. Reference to the generator is obtained through the input object and data is obtained as the return value of calls on any of the generator's processes.

Inputs can be created at any time. Creators of input objects must specify, at initialization time, which kind of generators "plug into" this input. To this end, they specify a list of classes from which generators must be derived in order to connect to an input. To ensure reliable connection to consumers, inputs are not permitted to be left "open": connections cannot be cancelled without a replacement. The semantics of an input are defined by the consumer-object to which it is attached and, in most cases, a consumer object is the creator of its inputs.

The generator-consumer pair, as obtained through multiple inheritance, is called a unit. Since most processing modules require signal and/or control inputs, most classes are subclasses of the unit-class. In a computation graph, generators that are not also consumers are referred to as sources and generators that are not also generators are called sinks. It should be noted that any number of sinks is allowed in a computation graph, representing simultaneous output through speakers, meters, files, and the like.

"k-rate" processing

Most computer music systems make use of multiple-rate processing as an optimization technique. The idea is to reduce computation cycles by updating control signals at a rate (k-rate) which corresponds to the bandwidth of human gesture rather than that of hearing.

As conditions remain static during a k-rate period, audio processing can be accomplished in blocks and will therefore execute faster on general purpose computers, under normal conditions.

Frequency-domain processing requires signal analysis, like the Fourier-transform, to be performed on minireal temporal "windows", extracted from the time-domain signal. Facilitating conceptual simplicity, frequency-domain units process one window per block (i.e., the analysis window advances in temporal intervals which correspond to the ratio of sampling-rate to blocksize).

The handicap posed by k-rate processing is the constraint it places on cyclic signal processing graphs; blocksize determines a minimum delay which occurs in recursive structures. Therefore, recursive signal processing will generally have to be carried out within a unit. This hampers the granularity of distributed computation which can be achieved, for example in physical modelling applications. (It should be noted that this restriction applies only to audio-rate processing but not to k-rate processing).

Hierarchical time

Computer music systems differ in the way they treat time. Apart from systems that handle time implicitly, a variety of approaches can be found. Music-n languages traditionally provide a "score" level (e.g., CMusic) or interfaces to algorithmic composition environments (e.g., Common Music) to handle time explicitly. Some systems address time-varying beyond deferred action (e.g., Nyquist, various implementations of the phase- vocoder and related techniques). In the design proposed here, a logical thread of time is represented by the timethread class. The timethread handles synchronization in a directed graph of generators and consumers; it represents a common clock for its set of subordinate objects and provides a common audio and control sampling rate. The impulse for block computation is a signal processing graph initiated by a driver object and propagated through the associated timethread. Subclasses of both the consumer and driver classes represent signal sinks which trigger calculation when a new block of data is required; this is illustrated by the speaker class, which interfaces to the audio output-facility, provided by the operating system.

The SubthreadDriver inherits from generator and driver and is used to connect timethreads in a hierarchical manner. It triggers block calculation in the slave timethread, as a function of the requests it receives from its own timethread. Specializations of the subthreadDriver are used to synchronize timethreads, which differ in blocksize and sampling-rate, by means of sampling-rate-convolution: they are also used as schedulers which defer invocation of sub-timethreads until necessary.

Signals processing is achieved in generators and consumers, under the assumption that time is moving toward in blocksize increments. Random access to positions in time require memory. The system provides the timewarp unit, which allows random access to a signal's memory representation. Timewarp can be used to realize arbitrary paths through stored sounds for use in granular synthesis, phase-vocoding and related techniques.

Implementation issues

The preliminary implementation of the system described above is based on Xerox PARC's Inter-Language-Unification (ILU) distributed object system. This public domain product currently supports and unifies implementations in C, C++, Common Lisp, Scltk, Python and Modula-3 in UNIX, Windows and

ICMC Proceedings 1996 255

Belbes & Lundén
Concurrency and data flow

Concurrency is handled following a very simplistic scheme in our implementation. In ILU, methods can be declared asynchronous; remote calls on asynchronous methods return immediately, without regard of method execution on the receiver side. Asynchronous methods are used in our implementation to pass tokens along a computational graph. A unit receives tokens that announce availability of data from every generator that feeds it; when all generators are available, it performs its block-computation and notifies each dependent generator upon completion by sending a token. To launch block-computation in a processing graph, a timeThread injects a token in the "pure" generators (sources), and the impulse is propagated through the graph as computation proceeds. No explicit mechanism like semaphores or lock-variables has to be provided for synchronous operation, because method calls are atomic by default.

Discussion

In this work, we have not addressed higher layers of application interworking. We are working on a unified approach to data storage and retrieval that will help in the organization of complex musical projects. Another related issue we intend to work on is portable high-level programming of graphical user interfaces. The kind of unification that has been demanded will help to maximize exploitation of programmers' efforts by relieving them from re-implementation of standard techniques and by reusing reuse of existing code. It will eventually support art-in-users in learning to handle complex computer music systems because concepts known from one application can be applied to others. It will also increase efficiency and refinement of artistic work with computer music systems in taking away the pains of hardware and software compatibility problems.

References


